Magistère informatique IM2AG 2015

Le magistère est une option pour les étudiants de L3 à M2 souhaitant avoir de l'expérience dans le domaine de la recherche.

Le 27 et 28 venez assister à la présentation de leurs stages contenant des sujets très divers et variés autour de l'informatique.

Programme :

Jeudi 27/08 :

- 9h - Lea Albert : Simulation parallèle de trajectoires sur multi-cœur et accélérateurs
- 9h45 - Thomas Baumela
- 10h45 - Rémy Boutonnet : Améliorations de l'analyse de programmes par interprétation abstraite
- 11h30 - Raphaël Jakse : Vérification de propriété à l'exécution avec un débogueur
- 13h30 - Luc Libralesso
- 14h15 - Lina Marsso
- 15h15 - Quentin Ricard
- 16h – Jules Lefrère : Humanité Digitale : Soutiens aux chercheurs helléniste et Latiniste

Vendredi 28/08 :

- 9h - Simon Moura
- 9h45 - Hugo Guiroux
- 10h45 - Antoine Faravelon
- 11h30 - Valentin Prulière : Techniques d'interaction en réalité augmentée mobile
- 12h15 Claude Goubet : Détection de vulnérabilités logicielles : Trouver des fonctions génératrices de débordement de buffer

Université Joseph Fourier GRENOBLE

UFR IM²AG UJF

UNIVERSITÉ GRENOBLE ALPES

Parallel simulation of trajectories on multi-cores or accelerators

Lea Albert
Supervised by : Florence Perronnin, and Guillaume Huard, and Jean-Marc Vincent

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

This article speak about the simulation of trajectory. It possible to simulate a trajectory with a sequential algorithm. The problem of this version is, it is too expensive : The backing of the trajectory is long.
One of solution to resolve this problem is to implement a parallel algorithm. Some versions exist but have problems with the task distribution and the trajectory backing.
My aim is to implement here an other version allow to resolve these problems and to reduce the execution cost greatly.

1 Introduction

1.1 Context

To estimate the performance of high system and network, it's necessary to simulate a long time Markov models.
This simulation allows to analyze the behaviour of some system, to capture scarce event sequences, to estimate coverage rates and to recover others information.

In the first part, we will speak about the classic simulation : Generate randomly an event (depending of probability model) and from a transition function, update the current state. This version is not too complicated but because of the long trajectory the execution time is too important.

One of solution to reduce the execution cost is the parallelization[**?**].
What we will talk about in the second part, by deviding the trajectory in intervals, we can compute each part in parallel.
There are some versions of parallels algorithms. One of them was made, called *Catch me if you can*[**?**] but has some problems of tasks distribution, and drop of efficient .
So, we want to implement an algorithm where the cost of trajectory backing are depreciated and where the cores are well allocate. It is we will discuss in the third paragraph.

Algorithme 1 : Algorithme séquentiel

Données : x_0 (état initial), T
pour t ***de*** 1 ***à*** T **faire**
  $e_t \leftarrow genere\_evenement()$;
  $x_t \leftarrow \phi(x_{t-1}, e_t)$;

Figure 1: The sequential algorithm of simulation [**?**]

2 The sequential algorithm

First at all, we will explain what is a Markov model to understand the simulation. A Markov model is a state machine without memory. It composed of states, events and transition function.
The sequential algorithm is, from a begin state, we generate the next state of the trajectory until the certain time(fig. **??**). All future states are computed thanks to the transition function, the current event and the current state.
A sequential implementation take long time with a important trajectory and present problem to store the final trajectory.
We want to reduce this cost by using all cores of the machine.

3 The parallel algorithm

The sequential version is too expensive to simulate a long trajectory.
To reduce the cost, we want to use a parallel version. Some piece of trajectory will be compute in parallel to produce the final trajectory.

Before continuing, it is important to introduce the notion of consistency.
A fragment of trajectory is called consistent if it is a part of the final trajectory. More, when a fragment trajectory crosses a consistent trajectory, it becomes consistent.

3.1 The algorithm of Nicol and Al.

[**?**] This algorithm is a parallel version to simulate a trajectory.

To begin, we have a trajectory of length *T*.
We will devide the trajectory in part and each core will be

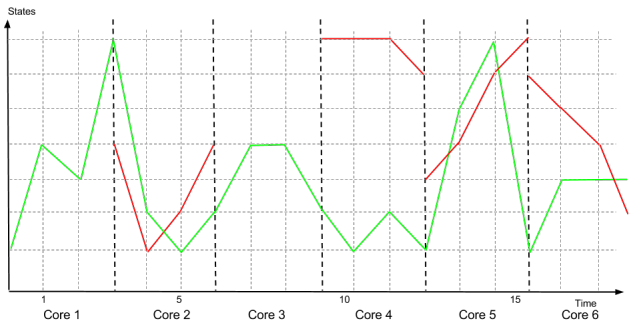


Figure 2: The Nicol and Al's algorithm

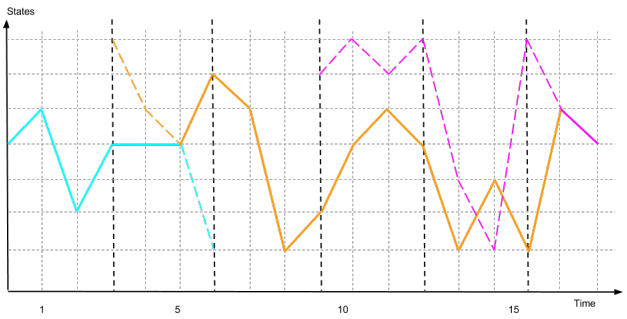


Figure 3: The Catch Me if You Can algorithm

work in the same time on interval.
For sample, we want to execute the simulation with 6 cores (fig. **??**). The trajectory length is 17. The core 1 will be work on the fragment [0, 3], the core 2 [3, 6], etc...
The simulation setting specify what is the begin state, so we know where the core 1 begin. We can say that the trajectory of the core 1 is consistent. The problem is for the others cores.
The method is for all cores (expect the first), we take randomly one state according to events. If the final state of a core *i* is different of the begin state of the core *i-1*, the core *i* starts again its work. For sample, in the fig. **??**, the core 2 begin with a different state than the final state of the core 1. It must to compute again.
We can note here that in the worst case the cost will be the same of a cost of an non parallel execution, so *T*.
The performance decreases because at the moment where two cores must to verify if the final state of one is the same of the start state of the other, they must to communicate. This say at the memory level a synchronisation which need time.
More, for storing the trajectory take also extra time.

3.2 Catch me if you can

[**?**] This algorithm is an improvement of Nicol and Al's implementation.

The principle is the same of the Nicol and Al's algorithm.
There is a trajectory of length *T* and we have *X* cores. There will compute in parallel the intervals.
The difference is, when two trajectories meet, they will be mixed up (fig. **??**). The meeting between trajectories does not occur at each front of interval but less frequently. This allow to cushion the cost of memory access.
As we can see on the figure **??**, the core 2 started from an other state of the core 1. We note that the core 1 go on to work and meet the trajectory of the core 2. It is at this time that the core 1 stop to work. The trajectory of the core 2 becomes consistent.
The problem with the implementation has been made is the improvements are not seen beyond three cores.
It's important to specify all cores share the same sequence of event. So, there is too many using of global variables (it must to use locks when a core wants to access it) and it causes extra time.
Furthermore, there is a task unbalance when a core is consistent, it stops. To finish, this implementation utilizes structures not very effective.
All of these problems don't resolve the problem of synchronisation, trajectory backing and best efficiency.
These algorithms present synchronisation problems or redundant computing.

4 Tools

Thanks to the monitoring tools (Oprofile or Likwid), I can analyze the *Catch me if you can* or my implementation.
This tools give me information about the memory : the cache misses, the CPU, the performance counters etc... We can get full information on the hardware during the execution of the program.

5 Contribution

5.1 The work performed

First at all, I compare the execution time of this algorithm with the sequential version. By testing on a set of 16 test, I notes that for 31%, the parallel implementation generated segmentation errors. In addition for 80% the sequential version is more effective. After this testing, I decided to see the code and to search the reason of these results.

As I said before, the first thing we noted is the high using of the global variables.
The simulation has parameters such as the number of threads, the trajectory size, the interval size and all cores must access at all these information. The problem is this requires the installation of locks.
The second thing that we observe is the simulation core was directly in the main function without separation on functions. There is no modularity in the code and if we want to implement others algorithms, we need to modify all the simulation part.

Thanks to this rewriting, following this simple analysis, I decided to implement a first draft to clean up the code. I leave out the simulation core for later.
I have seen better the choice of the author. Some variables

Figure 4: The new parallel version

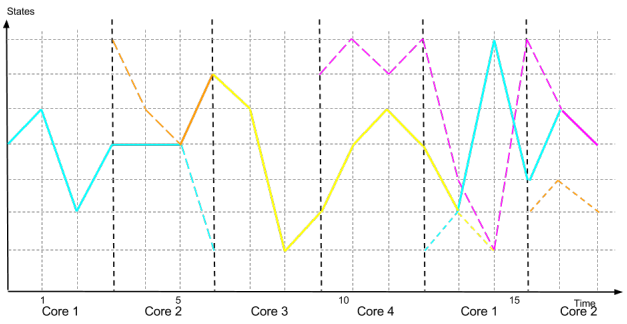


Figure 5: A parallel algorithm of simulation

must to stay global due to the using of PSI's library. For the rest of variable that I can remove, I put them in intern. If we need them, just put them in settings.

And then to made a code more clean and readable, I did structures. It composed for sample of all information useful for the settings of the simulation (the start states, the seed, the stop function, the trajectory size ...).
Then, I decided to implement a new parallel version : **??**

The trajectory is cut in different part and each part work on a fragment. There are more fragment than cores. This allow to resolve the problem of task unbalance : when a core finish its can work in the other fragment.
The only trajectory to be consistent is the first core.
When an other trajectory of a core i meet the consistent trajectory, the first core copy all states compute by the core i and the core i now must to compute an other fragment of the trajectory.
If a core has spent too much time to calculate the same interval without ever becoming consistent, so it calculates an other interval.
The implementation focus essentially on the problem of task distribution.

5.2 Improvements

Thereafter, I want to test my implementation, see thanks to profilers, performances loss of points. I want to improve the code, I want to identify structures which are ineffective and understand how reduce the cost of the backing trajectory. For this, I want to try to grow the intervals where we check the core trajectory or an other possibility is to store less information. Also, I want to implement others algorithms and see if there are differences of performances.

6 Conclusion

The simulation of a long trajectory takes an important time. For reduce this cost, one of solution is to implement a parallel algorithm.
Catch me if you can is a parallel implementation but present some problems. It wasn't performance compare of the sequential version. That's why I clean up the code in the first past to eliminate some performance looses.

By improve the number of fragment, so the amount of work, we can resolve the task distribution problem. Others implementations will be interesting to try and test.

Software encapsulation of hardware IPs
Magistère en Informatique de Grenoble 2015

Thomas Baumela
Supervised by : Pr. Olivier Gruber and Pr. Frédéric Pétrot

I understand what plagiarism entails and I declare that this report is my own, original work.
Thomas Baumela, August 2015

Abstract

Recent studies show that 70% of Linux Kernel crashes are caused by device driver bugs. This is explained by several facts: device drivers are hard to write and new device drivers are needed all the time. Device drivers are hard to write because they are highly dependent on both the specifics of each operating system and the minute details of hardware devices. Furthermore, they are often executed in privileged mode, hence a bug in a driver usually means a system crash.

To address these problems, we advocate the following solution: split-drivers as services interconnected through a software bus. With a split-driver, the complex hardware related part of a driver, called the back-end part of the driver, can be executed by the device, assuming that devices embed a small processor—something that is increasingly true with today's complex IPs. The other part of a driver, called the front-end, becomes a simple class-generic driver that runs as a regular driver in Linux. The two parts, the front-end and back-end, communicate using our software bus, exchanging messages.

Besides encapsulating bugs on the device itself, allowing to simply reset the device if the back-end driver was to fail, the proposed approach is quite important from a business perspective. Device manufacturers only need to write a single device driver. Not only they have the skills needed, but since the back-end driver runs on the device, they fully control and master the hardware/software environment. Furthermore, shipping a device means that it is readily available, no need to wait for the next Linux update cycle to push the new driver.

1 Introduction

Writing a device driver is hard, some say it is black magic. Writing a device driver requires a very detailed understanding of the kernel hosting that driver, the Linux kernel in our case. But writing a device driver also requires a perfect understanding of the hardware device itself. Not only writing a device driver is hard, but buggy device drivers usually mean an unstable machine and most often a brutal kernel crash. Of course, device drivers become more and more stable over time, like any other software, but hardware devices constantly change and new devices appear all the time. Writing device drivers and getting it right is therefore a never-ending struggle.

Since the 1960's device drivers are based on writing and reading values into device registers. But in contrast to user-space programming, writing and reading regular memory locations, reading and writing device registers are about communicating with the device, following a communication protocol that is specified for each device. For instance, this implies that some of these operations need to be performed in a particular order and using a precise timing. The necessary information is of course available in the hardware documentation, if the device manufacturer provides it, otherwise, retro-engineering is necessary. Even when available, hardware specifications are complex and notoriously incomplete from the perspective of device driver writers.

Device drivers are often provided by the manufacturer itself, who deeply knows the behavior of its device. However, the challenge becomes the required knowledge of the hosting operating systems, such as Windows, Mac-OS, or Linux. In the embedded world, many other operating systems exist, such as WindRiver, eCos, or variants of Linux like uclinux. In the Linux kernel, device drivers are written as kernel modules. A kernel module is a program that is dynamically loaded and unloaded into the kernel at runtime. Writing kernel modules requires a non-negligible investment and a good overall understanding of the Linux kernel. For more information on Linux device drivers, please refer to this book [Corbet Jonathan, 2005]. Few developers in hardware shops have this know-how and therefore drivers remain hard to write.

Our idea to solve these issues is to use the concept of split-drivers, currently used by Xen [Chisnall, 2007] with para-virtualized Linux images. The concept of a split-driver is simple. Encapsulate all the hardware specifics in the back-end driver that executes on the device. Use front-end drivers that are generic drivers for different classes of hardware devices, such as mouse, keyboard, or storage. The front-end

and back-end drivers communicate through our software bus. Each driver is a service, registered or unregistered from the bus, following a hot-plug model. Once registered, the front-end and back-end drivers communicate through messages.

The message-oriented protocol is generic, related to the class of devices, no longer about the hardware specifics of a given device. Each class of devices, such as a mouse or a disk, only needs to define one generic protocol for the generic front-end driver to communicate with any back-end driver of that class. This is somehow generalizing the popular USB-style split-drivers. This approach is especially interesting for System on Chip (SoC) that are increasingly moving towards a distributed architecture where hardware devices are accessed through a Network-on-Chip (NoC). It would make then sense to split all drivers, all relying on the use of our software bus, interconnecting front-ends and back-ends across the NoC.

This document is structured as follows. First, we will describe the software bus interface and how it can be used to set up multiple communicating software modules, from the user perspective. Then we will define the design of our software, how it internally works, the protocols and corresponding automata. Finally we will give the details of the Linux integration, in a simulation environment.

2 Software Bus

In this section, we will present the software bus API and the programming concepts from the user perspective.

2.1 Broker API

At the heart of our software bus is the broker, a service-oriented registry interconnecting front-ends and back-ends. Front-ends and back-ends are services that can dynamically register and unregister. Once registered, services can establish channels in order communicate through messages. Consequently, services are written following an event-driven programming style.

The API is object-oriented in style with entities defined through a state (C structure) and an interface (function pointers). The broker interface is given below. The broker functions are called by services to register and connect. Notice the broker is a singleton accessed through a global variable (gBroker). Notice also that all broker functions are asynchronous.

```
struct service_broker_i {
    status_t (*register_service) (struct
        service_broker* this,
            struct service* service);
    status_t (*unregister_service)(struct
        service_broker* this,
            struct service* service);
    status_t (*track_service) (struct
        service_broker* this,
            struct service_tracker* tracker);
    status_t (*untrack_service)(struct
        service_broker* this,
            struct service_tracker* tracker);
    status_t (*post)(struct service_broker*
        this,
            struct service* serv, void* evt);
};
struct service_broker {
    struct service_broker_i* ops;
    char* name;
};
extern struct service_broker* gBroker;
```

The service state and interface are given below. Note that a service is described by a name and properties so that it can be tracked. The callbacks are used by the broker to notify the service of various events. For instance, notice the two notifications confirming that a service is registered or unregistered. Notice the react function for reacting to self-posted events (via the broker post function). Finally, notice the connected function to be notified that a communication channel is established from the described service.

```
struct service_props {
    uint32_t service_id;
    uint32_t vendor_id;
    char     name[MAX_NAME_LENGTH];
};
struct service_i {
    /* React to a self posted event. */
    status_t (*react)(struct service* this,
        void* evt);
    status_t (*connected)(struct service*
        this,
            struct service_channel* channel,
                struct service_props* props);
    status_t (*registered)(struct service
        *this);
    status_t (*unregistered)(struct service
        *this);
};
struct service {
    struct service_i*     ops;
    const char*           name;
    struct service_props* props;
};
```

A registered service can track other services through tracker objects. The concept is that a tracker will be presented with registered services and asked if they are a match. If they are a match, the broker will establish a communication channel.

```
struct service_tracker_i {
    status_t (*match)(struct
        service_tracker* this,
            struct service_props* props);
    status_t (*connected)(struct
        service_tracker* this,
            struct service_channel* channel,
                struct service_props* props);
};
struct service_tracker {
    struct service_tracker_i* ops;
    struct service*           service; //
        tracking service
};
```

Channels are the mean for services to communicate by posting events. There are two sides to a channel, the tracking side and the tracked side. This means that there are two

channel objects, created by the broker, that permit posting events to the other side.

```
struct service_channel_i {
    status_t (*close)(struct
        service_channel* this);
    status_t (*post) (struct
        service_channel* this, int evt,
        void* data);
};
enum channel_kind {
    TRACKING_SIDE,
    TRACKED_SIDE
};
struct service_channel {
    struct service_channel_i* ops;
    enum channel_kind        kind;
    const struct service*    service;
    struct service_channel_listener*
        listener;
};
```

To follow what is happening on a channel, a service creates a channel listener. A channel listener mainly reacts to events posted by the other side of the channel. A channel listener is also informed when the channel is disconnected.

```
struct service_channel_listener_i {
    status_t (*react)(struct
        service_channel_listener* this, int
        evt, void*
data);
    status_t (*disconnected)(struct
        service_channel_listener* this);
};
struct service_channel_listener {
    struct service_channel_listener_i* ops;
    struct service_channel*
        channel;
};
```

2.2 Standard service interraction

Figure 1 illustrates the standard service interractions via our Broker :

- Registering a service
- Tracking a service
- Establish a communication channel
- Exchange messages
- Closing the channel

Consider two services A and B. The goal of A is to connect to B and to send it a message. Each service needs to register with the broker, using the *register_service* function (1). When the registration is done, A has been added to the known active services and it receives a *registered* notification (2) that confirms that it is now registered.

Because A is looking for B, it needs to track it via our broker. A's *Tracker* consists of several callback functions, but the important one is the *match* callback, asking the tracking service if another service is matching its particular interest. In the particular case of our service A, the *match* callback will

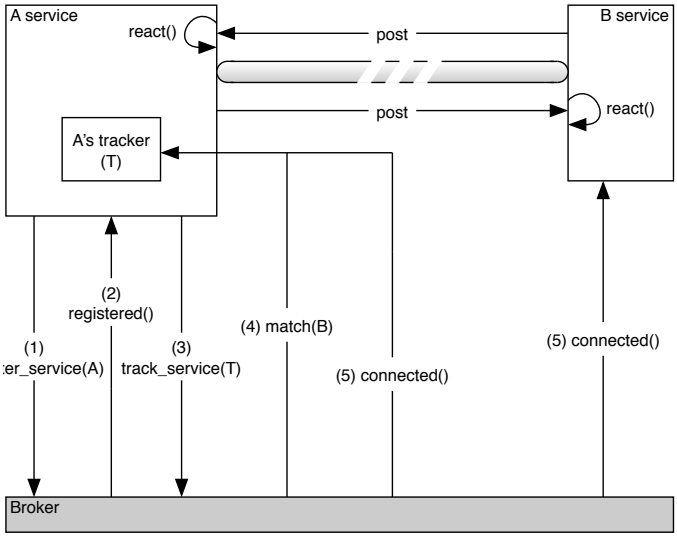


Figure 1: Example scenario of how broker can be used

return true only when asked if the service B is a match. Since our broker knows that A is tracking a service, it will call the tracker's *match* callback of A (4) for every already registered service (other than A, of course). As new services are registered, our broker will keep invoking service A for a match. When the service A declares a match on the service B, our Broker establishes a connection between A and B through a *channel* and notifies both services (5) that the connection has been established.

The service A then sends a message to B through the channel using the *post* channel function. The Broker calls the *react* callback on the service B. Communication will continue until either side closes the channel.

3 Software bus design

The design of our software bus has been influenced by our assumptions about the hardware. This hardware architecture is composed of a host system and multiple IP systems. These systems are connected to a NoC and a Network Interface (NI) is attached to each IP systems. A Network Interface is a piece of hardware that handles message transmission between IPs and host system. When a message needs to be sent from an IP to the host, the software writes the message in the local memory. The NI then reads the message and transmit it through the NoC to finally write it into the host local memory. Symmetrically, a NI can transmit messages from the host system to its IP system.

Software bus instances are running on each IP systems CPUs and on the host system CPUs. Each instance is exactly the same as the design of the software bus is symmetric. When a service is registering itself, it is effectively registered on the local instance running on the same CPU. To perform the service interactions seen before (channel creation, message posting, etc.) broker instances need to exchange control events, using the NoC. These control messages are used by the protocols we will now introduce.

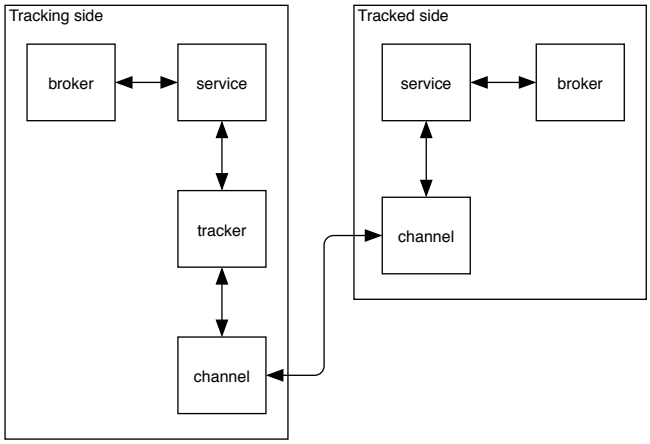


Figure 2: Broker data structures

Several elements are influencing our choices for the protocols between bus instances. First, they need to perform the expected behavior of the software bus, calling the right callbacks at the right time, in the right order. Also, they need to ensure that at every time, a data structure reference, sent by a software bus instance to another, is meaningful. This means that we need to be careful about when data structures are disposed. Figure 2 shows these data structures and their reference relations.

Registering a service: Figure 5 shows the typical steps to establish a communication channel : Registering a service, tracking a service and channel communication.

When a service register itself on a broker instance, a *broker_service* structure is allocated, describing the registered service structure. This structure is added to a list of registered services and a control event is broadcast to all broker instances, notifying that the new service is registered. Other instances will be able to allowing their services to track this new service or simply ignore the control event. The control event contains the new *broker_service* structure pointer and the service properties. Properties can be used at any time but broker_service pointer act as an unique identifier and should only be dereferenced on the broker instance that creates it.

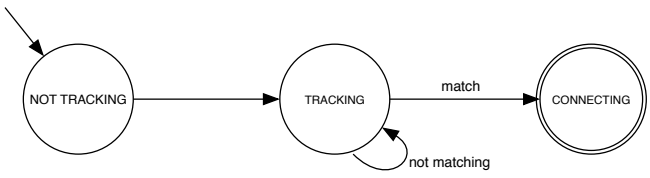


Figure 3: Service tracking automaton

Tracking a service: When a service tracks another one, the local broker instance sends a control event to all other broker instances, asking them to send all their registered service properties (and service pointers used as identifiers). The tracking service's broker instance receives these properties and call the tracker's match function for each of them. When there's a match, a channel connection is initialized. Figure 3 shows the service tracking automaton.

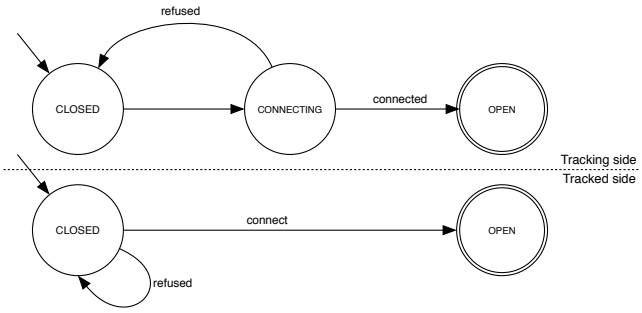


Figure 4: Channel creation automaton

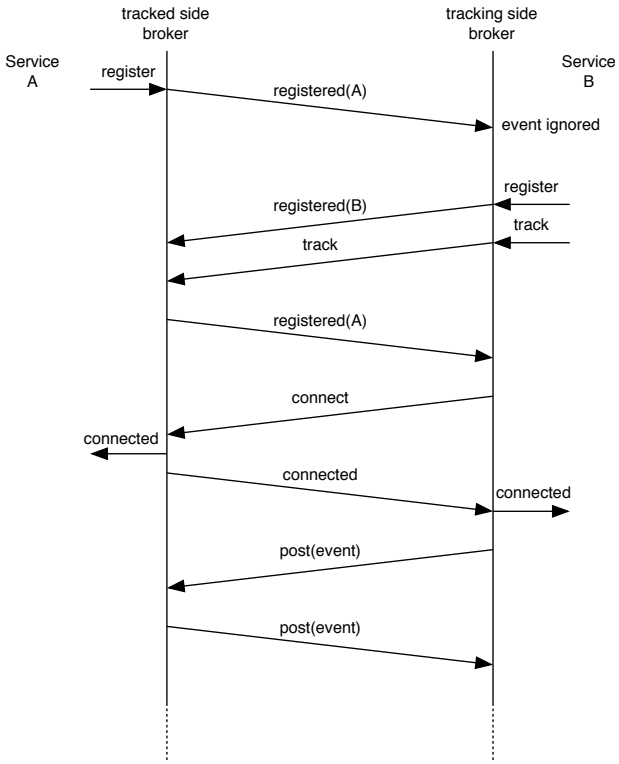


Figure 5: Classic interservice communication

Channel connection: Channel creation is always initiated tracking side. When a broker instance wants to create a channel, it first allocates the channel data structures for its side. Then it sends a channel creation control event to the broker instance where the tracked service is registered. This control event contains a channel pointer used to connect the two

sides, the tracker pointer and the tracking service properties. The tracked-side broker then creates its channel structures and notify the service that it is connected. It then sends an acknowledgment control event, containing its channel pointer. Finally, the tracking side broker complete its data structures and notify the tracker that it is connected. Figure 4 shows the channel creation automaton.

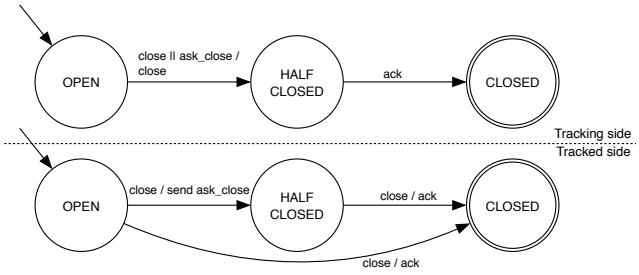


Figure 6: Channel disconnection automaton

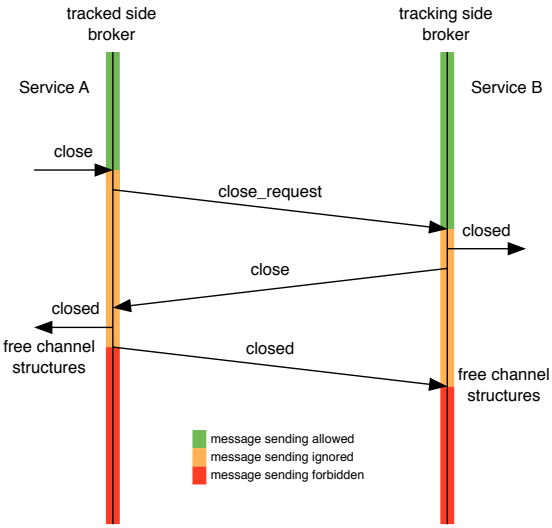


Figure 7: Channel disconnection protocol

Channel disconnection: Channel disconnection is again always initiated by the tracking side. If the tracked service wants to disconnect its channel, its instance sends a control event asking the tracking-side broker to start the disconnection protocol. Tracking-side broker sends a disconnect control event to the tracked side (after changing the channel status). The tracked-side broker completely removes the channel (and free memory) and notifies the tracked service that the channel is closed. It then sends an acknowledgement control event to the tracker-side broker that will free the channel data structures. Figure 6 shows the channel disconnection automaton. Figure 7 illustrates the channel disconnection protocol.

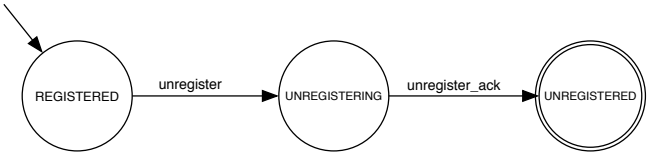


Figure 8: Service unregisteration automaton

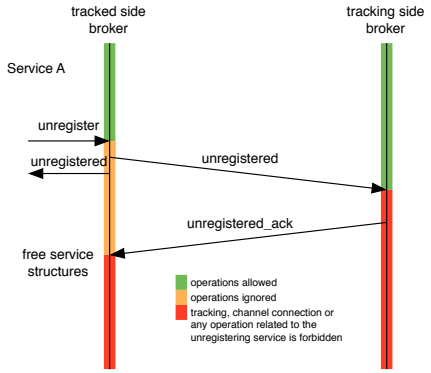


Figure 9: Unregistering service protocol

Unregister service: To unregister a service, the local broker changes the service status to unregistering and broadcast an event to all other instances. The other instances must answer by sending back an acknowledgement event. When all acknowledgments have been received, the local broker knows it is now safe to free the service data structure. Figure 8 shows the service unregistration automaton. Figure 9 illustrates the corresponding protocol.

4 Linux Integration

The integration of our software bus in Linux was a crucial step. We are happy to report that the integration is a success, even if more testing is still in order. In this section, we will first briefly recall the basics of device drivers in the Linux kernel and then discuss how we integrated our software bus. Finally we will sum-up all the software and simulated hardware prototypes that has been developed and tested as proof of concepts.

4.1 Linux device drivers

A Linux device driver implements the following four methods: open, close, read and write. For example a block driver in Linux is attached to one or multiple block devices. When a system call is made on a block device, the call is redirected on the corresponding driver method. A device driver has essentially two primordial roles: (i) drive a specific hardware and (ii) manage the impedance mismatch between software and hardware programming models.

Driving a specific hardware is done through reading and writing device registers, interacting with the finite state automaton implemented in hardware. This requires in depth knowledge of the device internals, usually found in more

or less detail in the documentation provided by the manufacturer. By essence, the hardware programming model is event-oriented. The driver requests something, the hardware will execute the request asynchronously, usually indicating the completion of the request by raising a hardware interrupt.

In contrast, the Linux kernel is fundamentally thread-oriented, assuming synchronous operations. For instance, the open/close and read/write operations are synchronous operations from the perspective of the user-level thread executing them. For example, a read operation will block until the read data is available. It is therefore the responsibility of the device driver to bridge the software world of blocking threads and the hardware world of asynchronous events.

The boundary between these two styles appears where the device needs to block a system call from the user, waiting for hardware events. On the one hand, the user thread is blocked, using thread mechanisms (mutex, semaphores, wait-queues, etc.). On the other hand, interrupt handlers are called with hardware interrupts, waking up the blocked thread. With our broker, front-end drivers also use the same mix of thread and event programming.

4.2 Split-driver

The idea of the split-driver is to split a Linux driver at the boundary between the OS specific and the hardware specific part of a driver. The front-end runs the OS specific part. In Linux, it means that the front-end is interfaced the system calls. Because the system call functions are synchronous, the front-end blocks the thread that made the call, while it is communicating with the back-end. The back-end driver runs the hardware specific part. When a front-end request is received by the back-end, it communicates with the hardware and send back a response. It also can notify the back-end of some hardware event, sending messages containing the nature of the hardware events.

Figure 10 shows an example of the split-driver model. The front-end driver is interfaced with the operating system as a block driver. The back-end driver drives the hardware, a disk in our case. Both communicate using our software bus as front-end and back-end are broker services. The interface of these services is event-driven. Notice that the front-end follows a similar model as classic Linux device drivers, using both threading and event-driven programming. The back-end is fully event-driven.

Front-end and back-end typically run on different systems. With one front-end running on the host system and back-ends running on the device system. Sub-systems are connected to a Network on Chip (NoC). The software bus situates on top of that NoC, abstracting it for the broker services. Figure 11 shows the hardware architecture, several subsystems, the software bus, and the front-end and back-end drivers communicating using the software bus.

4.3 Software bus integration

For our experiments, we use a software functional simulation of this environment, allowing us to experiment our solution faster and easier than on a real hardware system. Our experiments are integrated in this simulation environment, written using the RABBITS environment developed by the TIMA

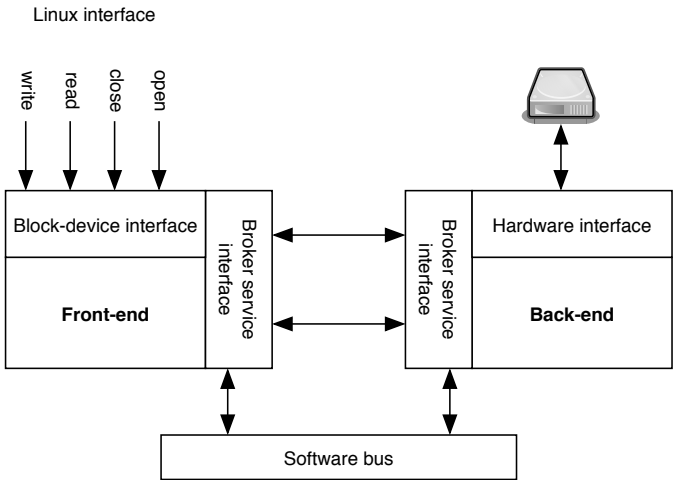


Figure 10: Split-driver model

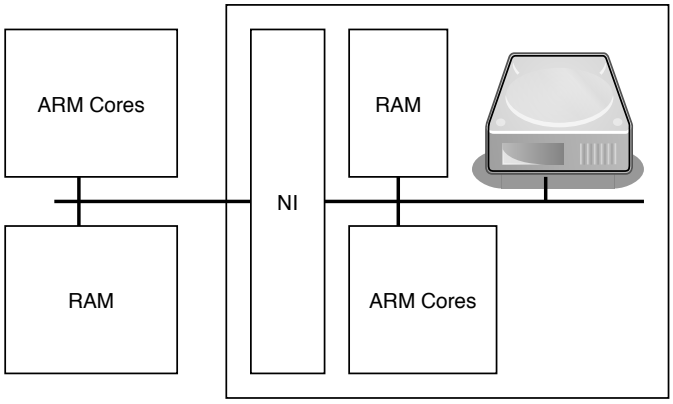


Figure 11: Hardware architecture

laboratory. The simulated hardware architecture (illustrated by Figure 11) is composed of a host system and an IP system. Both host and IP system have ARM cores, memory and a serial device used for console access. The IP system has also a Network Interface (NI) device, abstracting the IP system as a simple device connected to the host NoC.

The NI is a hardware element, written in a previous work and integrated as a RABBITS hardware component. The NI component holds FIFO buffers for receiving and sending messages, respecting their departure order. It is designed with ring buffer algorithm that allows to use lock free algorithm to send and receive messages. NIs reads and writes messages directly in the ram, using Direct Memory Access (DMA) into ring buffers that has been initialized by the software running on the systems.

To integrate our software bus, we first needed to write a simulation platform, in C++ language, using systemC and the RABBITS environment. This platform performs a functional simulation of the described hardware architecture. On both host and IP systems, a small custom Linux distribution is running, containing the minimal tools to loading kernel modules.

This solution is faster than developing on a real hardware system immediately and accurate enough to demonstrate the behavior of our software implementation.

Our broker implementation is wrapped into a kernel module, as device drivers are. This module is in charge both to perform the protocols introduced in the broker design section and also to use the NoC hardware interface to sends and receive data over the NoC. To send data overt the NoC, a broker instance drives the NI hardware. To send a message, the broker writes the message into the ring buffer and increment its write ring buffer index. The NI then throws back a hardware interrupt, notifying the broker that a message is on departure, and sends it to through the NoC. At the destination, the NI copy the message into the destination system's memory and notify the message arrival with a hardware interrupt.

To show the functional behavior of our software bus, we have developed a ramdisk split-driver. Back-ends are running on the IP system and contain all the data storage logic. The two ends communicates using our software bus, using their custom read/write request/response protocol. Several back-end can be dynamically loaded and unloaded. When a back-end is loaded, it registers a service into the software bus and tracks the front-end. When connected to the front-end, the front-end dynamically creates a device file into the host file system, allowing user to read or write into the device. When a read or write operation is required, the front-end sends a appropriated. The back-end process this request, by reading or writing data into its data storage, and then sends back a response containing either the required data in the case of a read operation, or an acknowledgment that the data has been successfully written. When a back-end is unloaded, the front-end, notified of the disconnection of a channel, dynamically removes the corresponding device from the host file system.

5 Conclusion

The work done so far is a proof of concept, demonstrating that our software bus can be implemented, integrated in Linux, and used to split-drivers. The software bus allows front-ends and back-ends distributed communications, taking place of the Network-on-Chip (NoC). The Linux kernels and our softwares are executed on a simulator of the real target platform.

The further steps will be to improve the Network Interface model in a continuation to improve its current behavior and interface, and to run on a real hardware, using an FPGA to model the SoC and NoC.

References

[Chisnall, 2007] David Chisnall. The Definitive Guide to the Xen Hypervisor. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[Corbet Jonathan, 2005] Kroah-Hartman Greg Corbet Jonathan, Rubini Alessandro. Linux Device Drivers, 3rd Edition. O'Reilly Media, Inc., 2005.

Verifying Good Practice Rules at Runtime with a Debugger

Raphaël JAKSE
Supervised by : Yliès Falcone, Kevin Pouget and Jean-François Méhaut
Laboratoire Informatique de Grenoble, UJF, CNRS, Inria, Corse

raphael.jakse@gmail.com, ylies.falcone@ujf-grenoble.fr,
Jean-Francois.Mehaut@imag.fr, kevin.pouget@imag.fr

Abstract

With the objective to help reducing the impact of software bugs, we define a way to ease discovery and understanding of a class of bugs related to good programming practices and good APIs or libraries usage. Our idea is to check properties relative to good practice and good usage at runtime using a debugger. In this paper, we extend the runtime property-checking model defined in a previous work and provide a efficient implementation of this extension. This extension is based on a trace slicing mechanism which allows efficient handling of parameterized properties, common among such good practice and good library usage properties.

1 Introduction

Writing softwares containing bugs is unfortunately common. A bug is a flaw preventing a system to behave as intended. By definition, eliminating bugs is desirable. Bugs can range from minor to critical. Minor bugs most likely cause annoyances while critical bugs can cost human lives.

One way of eliminating a software bug is to notice its existence by observing one of its consequences, find its cause in the source code and fix it. Fixing a bug is, in many cases, the easiest step of eliminating it when it is fully understood. Finding out its existence and fully understanding it, however, can both be really hard.

Bug discovery by manual testing. Several ways of discovering a bug exist. One of the most obvious ways is observing one of its consequences by using the software. Most obvious bugs can easily be spotted this way during the development of the software by testing manually modifications to the code or by a team responsible for testing the software Itkonen *et al.*. Bugs are also spotted by final users of the software, which, depending on the development model, the kind of software and the severity of the bug, is more or less undesirable.

Automatic testing. Automatic testing through unit tests can be used to ensure already fixed bug do not show again, to limit regressions and to check the code is correct for a restricted set of inputs Itkonen *et al.*. Research have been done on automatic generation of unit tests Cheon and Leavens Cohen *et al.*.

Preventing bugs with static analysis. Another common way of discovering bug is using static analysis or abstract interpretation Cousot and Cousot. With these methods, the source code of the software is analyzed without being run in order to find elements which are most likely programming errors or which might cause maintenance difficulties, raising the risk of introducing bugs during subsequent modifications. They can also prove some properties over the software's behavior. Unfortunately, while theses approaches are extremely valuable, they are not exempt of flaws: they can be really slow, limited to certain classes of bugs or properties, produce false positives and false negatives. While static analysis and abstract analysis can provide certain guaranties by proving properties over the program, proving correction of a software by static analysis is unsolvable in the general case Landi.

Theses approaches of finding bugs, as well as other approaches not covered in this introduction, have their own sets of drawbacks and benefits, making them all valuable for discovering different sets of bugs in different situations: some kind of bugs are more easily avoided by analyzing the software statically, other bugs are more easily and rapidly spotted by testing the software directly.

There are, however, bugs that are not discovered or understood easily or efficiently by aforementioned methods: as said above, static analysis cannot find every bug in all softwares and some bug are not apparent in an obvious way in all tests. When a malfunction shows during a test, its cause is not always easily understandable as the programming error at its root could be caused far before during the execution of the software (e.g. a pointer that should contain a valid address instead of a null or a random address).

Motivations. In this paper, we therefore present a technique complementing hereinabove methods, trying to gather their respective advantages, with a different set of benefits and drawbacks, thus providing a way of finding and understanding a class of bugs more efficiently and more easily in certain situations.

An entire class of bugs is caused by misuse of Applica-

tion Programming Interfaces (API) or libraries or by breaking common programming rules such as "do not modify a container while you are iterating on it, except under certain documented specific conditions". Such rules, as well as "good API/library usage", exist because if they are not followed, undefined or unattended behaviors can occur: while "iterating over a container in a given order" can be easily defined, "iterating over a muting container in a given order", under certain situations, can be hard, nay impossible to define, resulting in undefined behavior or runtime breakage. In the case of libraries or APIs, using them in a way that differs from the intended way can make the library user reach an untested and unforeseen case which can trigger bugs in the library or the application. The unintended use can also work, by chance, in the used version of the library, but the guarantee that it will work in a future version of the library is inexistent.

Goal. In this paper, we aim to present a method based on a debugger to verify, during the execution, "good practice" kind of rules, that is, rules inspired by common programming rules and properties describing correct API or library use.

We first introduce a simplified model inspired from the one we introduced in our previous work Jakse *et al.*. We then introduce an extension of this model to handle such properties efficiently. This extension is based on the concept of trace slicing as described and formalized by Roşu and Chen Rosu and Chen. We describe an efficient implementation of this extended model, used in our proof-of-concept.

2 Description of the Existing Property Checking Model

Debugger-based property checker. Properties described in this paper are checked over the execution traces of a software with the help of a debugger. The property checker is a whether a part of the debugger or an extension using a programming interface this debugger provides.

An execution trace is a sequence of events which can be monitored by a debugger, ordered by time. Properties are checked during the execution ("online"), meaning that while processing an event, the next event is not accessible. Theoretically, nothing prevents the trace from being infinite, though traces are always finite in practice and some properties might require the end of the trace to give a verdict.

Events which can be monitored by a debugger mainly include function calls. A function call event includes any value which can be accessed by the debugger before the first instruction of the function's body, including its parameters and the current value of global variables accessible from within the function. It also includes the return value of the function if, instead of considering the moment of the call, we consider the moment when the function returns. In practice, function calls are monitored through breakpoints. In order to monitor a function call, the property checker sets a breakpoint on the first instruction of the function's body and then asks the debugger for values needed to check the property. As an illustration for this model, see the architecture of our proof of concept in Figure 1, taken from our previous paper.

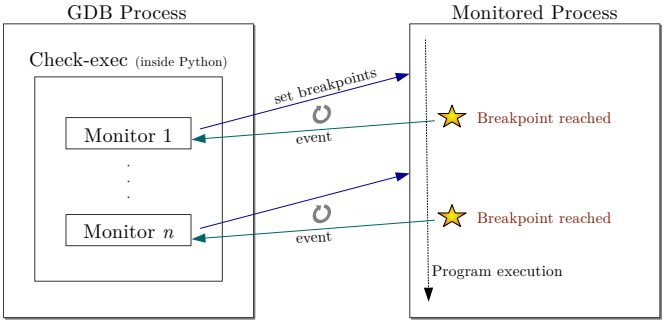


Figure 1: Architecture of `check-exec`, our proof-of-concept. GDB is the debugger on which we based our implementation. GDB provides a Python programming interface, we therefore chose to wrote our proof-of-concept in Python. A monitor is an instance of the property-checker. A monitor corresponds to one property.

Other events that can be monitored include syscalls and memory accesses (read, write or both). Syscalls can most likely be viewed as regular function calls in our model, though in practice monitoring syscalls involves different mechanisms such as catchpoints instead of breakpoints. Monitoring memory accesses, however, cause important performance issues in practice, thus needing more research in order to take advantage of this possibility. Both kind of events will be discussed in Section 4.1.

The advantages of basing the property checker on a debugger are discussed in our previous paper. They mainly consist in good performance, as the monitored program runs at native speed. Performance penalty are mostly caused by handling breakpoint and the time needed to change the property's state. Another advantage is that debuggers already are a familiar tool for developers so they do not need to learn a completely new tool to adopt this model. Checking properties at runtime, while limiting verification to tested executions, allows to handle more complex properties and do not require long code analyses, permitting to adopt the property checker in a fast modification-test loop based software development.

Automaton-based property description. In our previous paper, we defined a formalism to describe properties over execution traces. We designed this formalism with the idea that it should be efficient and feel as most natural as possible to developers, and be general or extensible enough to describe most properties. Therefore, we chosen to build a formalism inspired by state machines, which can be found in many computer science areas such as hardware design, human-computer interfaces and language theory. We here describe a simplified model which is sufficient for the needs of this paper but a more complex model is described in our previous paper, in which more details are given.

In this formalism, an automaton consists in a set of states Q which can be accepting or non accepting, a set of transitions δ, an initial state init and a initial variable environment σ_0 representing the initial memory of the automaton. A variable environment is a mapping from variable names to

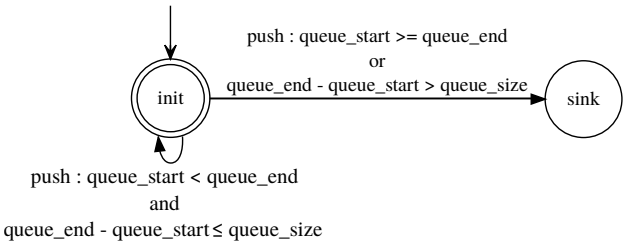


Figure 2: Example of a small property checking whether a queue overflows

variable values. The input symbols of the automaton are the events of an execution trace of the monitored program. See Figure 2 from our previous paper for a minimal example of an automaton in this formalism. In this example, the sink state is non-accepting while the init state is accepting. The init state is reached when a queue overflow occurs.

(q_k, σ_k) describes the internal state of the automaton before the k^{th} event of the trace, including its current state q_k and its current variable environment σ_k. The property described by the automaton is verified at the step k if and only if the current state is an accepting state.

A transition t consists in a start state $\text{start}(t)$, an event name, a guard $\text{guard}(t)$, an action and a destination state $\text{dest}(t)$. For each event concerned by the transition, the guard of the transition is a function depending on the values of the parameters values of the event and the current variable environment σ_k, returning a value in $\{\text{true}, \text{false}\}$. The guard decides if the transition must be taken. The action is also a function depending on the values of the parameters values of the event and the current variable environment σ_k, but contrary to the guard, it returns a new variable environment containing the same variable names as σ_k. At the beginning of the execution trace, the current state of the automaton is the initial state. For the k^{th} event e_k of the trace, transitions from the current state to a destination state describe the next internal state (q_{k+1}, σ_{k+1}) of the machine. For each transition t of the current state, if the event is concerned by the transition (the event name of the transition matches e_k), if the guard returns true, the new environment σ_{k+1} is given by the return value of the action and the new current state q_{k+1} is the destination state of the transition.

3 Extending the Model to Support Trace Slicing

3.1 Motivations

In this subsection, we briefly introduce trace slicing and we give examples of properties showing the need for trace slicing support in our model.

Many properties, including "good practice" and "good usage" properties, are to be checked on a per-object, or a per-family of object basis, rather than at a more global level. These parameterized properties are not to be checked against the entire execution trace. They are to be checked against each part (*slice*) of the execution trace specific to an instance of the object, or family of objects to which they relate. In the next paragraphs, we give examples of such parameterized properties.

Simple parameterized properties. The simplest parameterized properties are those involving one object.

First of such properties are those following the acquire-release resource model Rosu and Chen, in which each acquired resource must be released and no resource can be released without being acquired, represented bellow with an regular expression-like syntax:

$$\forall r, (\text{acquire}(r) \cdot \text{release}(r))^*$$

Or, in a negative way:

$$\nexists r, \text{begin} \cdot \text{release}(r) \,|\, \text{acquire}(r) \cdot \text{end}$$

The regular expression, in both cases, is not matched on the entire execution trace, but it is matched with every slice of the execution trace corresponding to an instance of the quantified parameter, that is, for each instance of r, the corresponding slice of the execution trace contains any event which does not involve any other instance of r.

Based on the same acquire-release model, any opened file should be closed:

$$\forall f, (f = \text{fopen}() \cdot \text{fclose}(f))^*$$

Reading from or writing to a file can only be done after it was open and before is was closed:

$$\forall f, (f = \text{fopen}() \cdot (fprintf(f)|fread(f))^* \cdot \text{fclose}(f))^*$$

Any allocated block memory should be freed:

$$\forall p, (p = \text{malloc}() \cdot \text{free}(p))^*$$

In C++, any created ("new"d) object should be deleted:

$$\forall obj, (p = new \cdot \text{delete}\, p)^*$$

Parameterized properties involving several parameters. There are also more complex properties involving a family of objects. This typically applies for object which are linked in one way or another. A good example of such family is collections and iterators. Meaningful iterators are always tied to a collection, and a corresponding rule is: an iterator must not be used after a modification of its corresponding collection if it was created before this modification. See Figure 3 for a description of this property with an automaton.

An example of slicing a fictive execution trace on this property is given in Table 1.

As there can be several iterators over the same collection, any event involving a collection without any particular iterator, like $\text{modify}(c)$, can concern several slices of the execution trace.

Slice	Parameter instances	Current state	Affected slices
Slice 1	(None, None)	init	
event: newCollection(c1)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(from slice 1)
event: newIter(c1, i10)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
Slice 3	(c1, i10)	iReady	(from slice 2)
event: next(i10)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	
Slice 3	(c1, i10)	iReady	(*)
event: newIter(c1, i11)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
Slice 3	(c1, i10)	iReady	
Slice 4	(c1, i11)	iReady	(from slice 2)
event: newCollection(c2)			
Slice 1	(None, None)	init	(*)
...	...	...	...
Slice 5	(c2, None)	cReady	(from slice 1)
event: modify(c1)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
Slice 3	(c1, i10)	invalidated	(*)
Slice 4	(c1, i11)	invalidated	(*)
Slice 5	(c2, None)	cReady	
event: newIter(c1, i12)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
...	...	...	...
Slice 6	(c1, i11)	iReady	(from slice 2)
event: newIter(c2, i20)			
Slice 1	(None, None)	init	
...	...	...	...
Slice 5	(c2, None)	cReady	(*)
Slice 6	(c1, i11)	iReady	
Slice 7	(c2, i20)	iReady	(from slice 5)
event: next(i10)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	
Slice 3	(c1, i10)	error	(*)
...	...	...	...
Slice 7	(c2, i20)	iReady	
event: next(i20)			
Slice 1	(None, None)	init	
...	...	...	...
Slice 7	(c2, i20)	iReady	(*)
event: modify(c2)			
Slice 1	(None, None)	init	
...	...	...	...
Slice 5	(c2, None)	cReady	(*)
Slice 6	(c1, i11)	iReady	
Slice 7	(c2, i20)	invalidated	(*)

Table 1: Trace slicing example with a property checking good usage of iterators and collections.

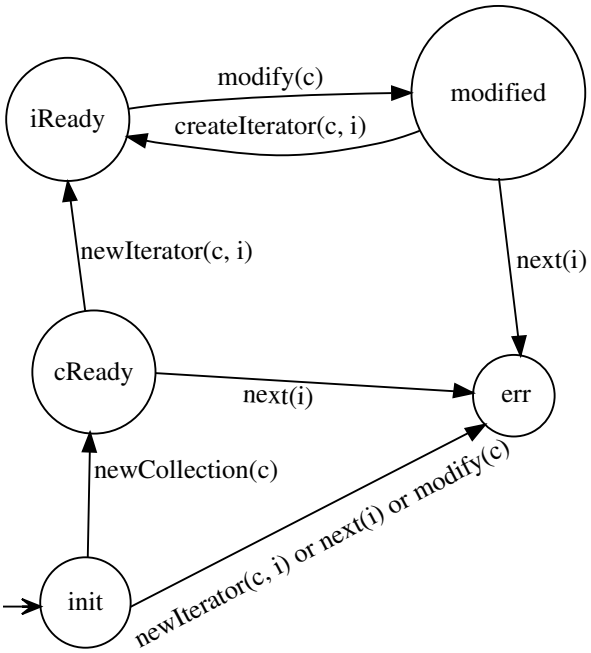


Figure 3: An iterator must not be used after a modification of its corresponding collection if it was created before this modification.

3.2 Slicing Automaton Model.

The existing model already handles parameterized properties like those presented in Section 3.1, being Turing-Complete thanks to the variable environment, the guard and the action functions of transition. However, it was designed for properties checking the state of a program at a global level. Writing parameterized properties in this model thus involves a lot of work that is duplicated for each such property and might even require "bypassing" the model by programming one in the automaton. The resulting automaton is thus not guaranteed to look natural and to be easy to understand. As such parameterized properties are common, we therefore define, in this subsection an extension to our model that allows to write parameterized properties efficiently and more naturally, while still handling unparameterized properties as a special case of parameterized properties with 0 parameters. Writing parameterized properties in this extension is similar to writing unparameterized properties in the base model, the only additional requirement being specifying the slicing parameters.

In the general case, though we know in advance how many parameters are involved in the trace slicing, as they are declared in the property with a mechanism similar to quantifiers, we do not know how many instances will be encountered while evaluating the property. We therefore need to be able to create slice dynamically during the evaluation.

Components of the slicing automaton. The slicing automaton model is defined by all elements of the base model, to which we add a tuple $S = (P_1, \ldots, P_m)$ containing the list of formal parameters on which the execution trace must be sliced.

$C_{k+1} \leftarrow \emptyset$
for all $s \in C_k$ **do**
  if $\exists t \in \delta, \text{start}(t) = q(s)$ and $\text{guard}(t)$ is verified **then**
    Let t be such transition
    $\sigma' \leftarrow \alpha_t(env(s), e_k)$
    if $\inf(I(e_k), I(s))$ **then**
      $s' \leftarrow (\text{dest}(t), I(s), \sigma')$
    else if $\text{comp}(C_k, I(e_k), I(s))$ **then**
      $C_{k+1} \leftarrow C_{k+1} \cup \{s\}$
      $s' \leftarrow (\text{dest}(t), \text{merge}(I(s), I(e_k)), \sigma')$
    else ▷ This slice is not concerned by the event
      $s' \leftarrow s$
    end if
  else ▷ This state is not concerned by the event
    $s' \leftarrow s$
  end if
  $C_{k+1} \leftarrow C_{k+1} \cup \{s'\}$
end for
[1]

Figure 4: Computing the next internal state of the slicing automaton

Before the event e_k, the current internal state C_k of the slicing automaton is a set of slices objects corresponding to the slices of the execution trace. Each slice object s contains a current state $q(s) \in Q$, a tuple $I(s)$ of parameter instances $(I(s)_1, \ldots, I(s)_m)$ and the variable environment $env(s)$. For any $i \in \{1, \ldots, m\}$, $I(s)_i = \text{None}$ if the parameter P_i of S is not instantiated in this slice.

The property described by the automaton is verified at the step k if and only if the current state of every slice of C_k is an accepting state.

Running the slicing automaton over a trace. At step $k = 0$, that is, before handling the first event, the internal current state C_0 of the slicing automaton is the singleton $\{s_0\}$ with s_0 such that $q(s_0) = \text{init}$, $env(s_0) = \sigma_0$ and $I(s_0) = (\text{None}, \ldots, \text{None})$.

That is, the initial internal current state consists in a single slice without any instantiated slicing parameter.

The internal state C_{k+1} of the slicing automaton at step $k+1$ is defined using the internal state C_k at step k and the k^{th} event, e_k, of the execution trace, according to the procedure 4.

We note:

- α_t the action function of the transition which, from a variable environment and the event, returns a new variable environment.
- $param(e)$ the function which maps a formal parameter of the event e to its instance in e.
- $I(e_k)$ the tuple of instances of the property's slicing parameters built from the parameter instances of the event e_k. If a slicing parameter instance is not given by the event, the corresponding component of the tuple is None.

The expression $\inf(i_1, i_2)$ checks whether each component of the parameter instance tuple i_1 is equal to $None$ or to the corresponding component of the parameter instance tuple i_2.

In the case i_1 is an instance tuple derived from an event and i_2 is an instance tuple of a slice, if $\inf(i_1, i_2)$ is true, we are in the case where the event belongs to the slice: every parameter instance given by the event is equal to the corresponding instance of the slice. If $\inf(i_1, i_2)$ is false, the event belongs to another existing slice or a slice which is to be created.

The expression $\text{comp}(C, I(e_k), I(s))$ checks whether e_k must lead to the creation of a new slice, given existing slices in C. This is the case if:

- $\forall i \in \{1, \ldots, \#(S)\}, I(s)_i \neq \text{None}$ and $I(e_k)_i \neq \text{None} \Rightarrow I(s)_i = I(e_k)_i$, that is, there are no incompatible parameters between parameter instances, and
- $\forall s' \in C, \inf(I(s), I(s'))$ is false or $I(e_k)$ and $I(s')$ are incompatible, that is, there is no slice at the same time more specific than s and compatible with the event parameter instances, "more suitable" for this event.

The expression $\text{merge}(I(s), I(e_k))$ gives a tuple of the same size of $I(s)$ and $I(e_k)$, in which the j^{th} component is equal to the j^{th} component of $I(s)$ if this component is not None, otherwise to the j^{th} component of $I(e_k)$, which can be None. If both components are not None, by construction of the procedure, they are equal. $\text{merge}(I(s), I(e_k))$ can be seen as a kind of least upper bound of $I(s)$ and $I(e_k)$.

4 Implementation of the Slicing Automaton Model

In this section, we describe an efficient implementation of the slicing automaton model defined in Subsection 3.2. A naive implementation would indeed be inefficient when dealing with execution trace containing a large number of instances of the slicing parameter tuple, as for each event of the trace, the set of slices would be entirely visited, and for each slice, this set can be visited a second time. Determining, at anytime, whether the property holds is also costly as it merely requires to loop over the set of slices to look for a non-accepting current state.

Our implementation, inspired by the algorithms described by Roşu and Chen Rosu and Chen and used in our proof-of-concept, make use of appropriate data "caching" structures to avoid looping over the set of slices altogether.

Optimizing access to the set of current states. In order to make this access efficient, we define `CSC`, a counter (multiset) which maps each $q \in Q$ to the number of slices of which q is the current state. Each time a slice is added, removed or updated, the `CSC` is updated. The multiset can be efficiently implemented as a hash table or, even better, by a simple array if states are represented by naturals. In both cases, obtaining the set of states means iterating over the set of states, listing each state q for which `CSC`[q] is not zero. If the set of state is expected to be large and the property often evaluated, a `curStates` array of $\#(Q)$ cells containing the list of current states and a `curStatesCount` natural giving the

number of current states can be maintained next to CSC, thus reducing the complexity of accessing the set of current states to a constant time.

Obtaining the set of related slices when an event is handled. The idea behind our implementation is to precompute the access of slices for future events when information needed to compute this access is easily accessible rather than when an event is handled. This information is easily accessible when we are creating new slices. At this time, it is possible to build the list of parameter instances tuples which are compatible with this slice and to connect these instances with this slice in a global map. These instances can contain wildcards for parameters which are not instantiated in this slice. When a event is handled, the parameter instance tuple is used to ask the global map to get the list of related slices.

We therefore define F, a table which maps a parameter instance tuple to a list of slices. The purpose of F is to be used as the global map described in the previous paragraph.

In Figure 5, we describe the procedure our implementation follows when an event is applied to a slice. $\mathrm{instance}(P, s)$ is the value of the formal parameter P in the slice s. If P is not instantiated in the slice $\mathrm{instance}(P, s) = \mathrm{None}$. Likewise, in In Figure 6, $\mathrm{instance}(P, e)$ is the value of the formal parameter P in the event e, or $None$ if e does not provide an instance of P.

The parameter compatSlices of the procedure APPLYEVENTSLICE contains slices which are compatible with the instances of the event's parameter instances. It is used in the expression $\mathrm{comp}(\mathrm{compatSlices}, I(e), I(s))$ in order to determine whether a more specific compatible slice exists. inf and comp have the same meaning as in the description of the slicing automaton model. However, in this implementation, we already built a list of compatible slices, so the implementation of comp does not need to check this again, only compatible slices are looped over, where a naive implementation could have made comp loop over the whole set of slices.

In Figure 6, we describe the procedure in charge of dispatching events to the right slices. The expression GETCOMPATIBLEF(p, s) builds, from the table F, the list of slices having instances p' which are less specific than p, where less specific means that each component of p' is whether equal to $None$, or to the corresponding component of p, that is to say, the slice s is added for all possible instance tuple that would be compatible with this slice.

In Figure 7, we give an implementation of GETCOMPATIBLEF which makes no assumptions on the internal implementation of the table F. Depending on the data structure used to represent F, this implementation can be optimized to reduce the number of operations. In particular, if F is implemented as a trie where components of tuples are used as its keys, prefix lookups can be gathered efficiently so the cost of inserting new tuple is reduced.

4.1 Limitations

Benchmark. The implementation described in Section 4 and the corresponding proof-of-concept still lack solid benchmarks to prove their efficiency. Unfortunately, we still need

```
1: procedure REGISTER(newQ, σ, i)
2:     create s
3:     dest(s) ← newQ
4:     env(s) ← σ
5:     I(s) ← i
6:     CSC[newQ] ← CSC[newQ] + 1)
7:     visitedEvents(s) ← eventId(newQ)
8:     let p be a tuple with #(S) components
9:     i ← 1
10:    for all P ∈ S in order do
11:        p[i] ← instance(P, s)
12:    end for
13:    ADD(F, p, s)
14: end procedure
15: procedure UPDATESLICE(s, newQ, σ)
16:    CSC[dest(s)] ← CSC[dest(s)] − 1
17:    CSC[newQ] ← CSC[newQ] + 1
18:    dest(s) ← newQ
19:    env(s) ← σ
20:    q(s) ← newQ
21: end procedure
22: procedure APPLYEVENTSLICE(s, e, compatSlices)
23:    if ∃t ∈ δ, start(t) = q(s) and guard(t) is verified
   then
24:        Let t be such transition
25:        σ' ← α_t(env(s), e)
26:        if inf(I(e), I(s)) then
27:            UPDATESLICE(s, (dest(t), σ'))
28:        else if comp(compatSlices, I(e), I(s)) then
29:            REGISTER (dest(t), merge(I(s), I(e)), σ')
30:        end if
31:    end if
32: end procedure
[1]
```

Figure 5: Computing the next internal state of the slicing automaton

```
1: procedure APPLYEVENT(e)
2:     let p be a tuple with #(S) components
3:     i ← 1
4:     for all P ∈ S in order do
5:         p[i] ← instance(P, e)
6:     end for
7:     compatSlices ← GETCOMPATIBLEF(F, p)
8:     for all s ∈ compatSlices do
9:         APPLYEVENTSLICE(s, e, compatSlices)
10:    end for
11: end procedure
[1]
```

Figure 6: Procedure to handle an event

```
1: procedure GETCOMPATIBLEFSTEP(p)
2:     if i < #(S) then
3:         res ← GETCOMPATIBLEFSTEP(p, s, i + 1)
4:         if i^th component of p is not None then
5:             p' ← p
6:             p'[i] ← None
7:             res ← res ∪ ADDCOMPATIBLEFSTEP(p', s, i)
8:         end if
9:     else
10:        res ← get(F, p) or default to ∅
11:    end if
12:    return res
13: end procedure
14: procedure GETCOMPATIBLEF(p)
15:    return GETCOMPATIBLEFSTEP(p, s, 0)
16: end procedure
[1]
```

Figure 7: Mapping a parameter instance tuple to all compatible slices

to build a decent set of good practice and good usage rules to be able to benchmark correctly. Moreover, an issue is affecting the performance of our proof-of-concept when setting a finish breakpoint, used to retrieve the return value of the function. Unfortunately, the return value often gives the addresses of objects to be checked, as well as failure codes which are also important elements. This issues needs to be worked out in order to measure performance correctly. However, we already have evidences that our optimizations are correct with our preliminary measures on toy properties[1].

Proofs. No proof has been written for algorithms given in Section 4 as of the writing of this paper. It is therefore unknown whether these algorithms are formally correct. However, based on previous proved work, any error in these algorithms should be recoverable.

5 Future Work

Monitoring memory. As mentioned in Section 2, monitoring access to memory in a debugger is likely to cause performance issues. Indeed, monitoring memory access is done using watchpoints. Watchpoints are efficient only if they are implemented by the hardware. Software watchpoints are slow, as they require checking whether monitored addresses in memory have been accesses after each assembly instruction. Unfortunately, in common CPU architectures, very few hardware watchpoints are available. In x86 architectures, a maximum of 4 hardware watchpoints can be set[2].

[1] we ran a test program opening and closing 1000 fake files along with a property-checker creating a slice for each such file. On a modest laptop, switching from a naive slice handling to a a slice handling using the optimizations described in the paper, we were able to divide the run time by nearly 10

[2] `https://sourceware.org/gdb/wiki/Internals%20Watchpoints`

This limitation reduces the set of objects which can be checked. For instance, C++ standard iterators are similar to pointers, making the property on iterators given in Section 3.1 more difficult to check. This property, however, is suitable for iterators provided by others libraries[3]. All the more, in languages such as C and C++, internal data of containers are often accessible to the developer, so it is possible to bypass containers' method, which can mislead a property checker which does not monitor memory. Further research is needed to address this issue.

Method and function overloading. The area of method and function overloading has not been explored yet in our model. This exploration is essential to make it useful in languages supporting overloading like C++.

6 Related Work

JavaMOP Chen and Roşu is the reference implementation by the authors of the trace slicing method used in the paper. As our implementation, it can be used to check (parameterized) properties at runtime. It applies to Java program and makes use of the monitoring features of the Java Virtual Machine, contrary to our model, which is based on a debugger and applies to a set of languages which is not restricted to the set of languages based on the JVM.

There is also a project aimed at checking good API usage. This project is SLAM and is restricted to system softwares, mainly drivers, on Windows. Unlike our work, it is based on static analyzing.

Valgrind is a framework to instrumentate binaries and check them for defects. It provides a way to detect memory related defects by a dynamic binary recompilation and instrumentation process of the sofware's machine code and by running it on a simulated CPU Nethercote and Seward. It provides a more comprehensive detection of memory related defects than our approach Nethercote and Seward, however our approach can be used to detect certain memory leaks more efficiently by writing a rule which checks that each manually allocated memory block is freed.

7 Conclusion

With this work, we introduce an extension to our existing model in the hope it will allow to build solid tools to help bug discovery and understanding relative to good programming practice. A solid set of properties taking advantage of this extension is yet to be built. We hope API designers and library writers will be willing to write rules for their products in order to help softwares using them be more reliable, with the additional benefits that these rules could be used as a complement to their documentation and usage examples.

References

[Chen and Roşu, 2007] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework.

[3] GLib, for instance, provide containers accessible through iterators. `https://developer.gnome.org/glib/stable/glib-Sequences.html`

In *ACM SIGPLAN Notices*, volume 42, pages 569–588. ACM, 2007.

[Cheon and Leavens, 2002] Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP 2002Object-Oriented Programming*, pages 231–255. Springer, 2002.

[Cohen *et al.*, 1996] David M Cohen, Siddhartha R Dalal, Jesse Parelius, and Gardner C Patton. The combinatorial design approach to automatic test generation. *IEEE software*, (5):83–88, 1996.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[Itkonen *et al.*, 2007] Juha Itkonen, Mika V Mäntylä, and Casper Lassenius. Defect detection efficiency: Test case based vs. exploratory testing. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 61–70. IEEE, 2007.

[Itkonen *et al.*, 2009] Juha Itkonen, Mika V Mantyla, and Casper Lassenius. How do testers do it? an exploratory study on manual testing practices. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 494–497. IEEE Computer Society, 2009.

[Jakse *et al.*, 2015] Raphaël Jakse, Yliès Falcone, Kevin Pouget, and Jean-François Méhaut. Verifying properties at runtime with a debugger. *(not published yet)*, 2015.

[Landi, 1992] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[Nethercote and Seward, 2007a] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

[Nethercote and Seward, 2007b] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[Rosu and Chen, 2011] Grigore Rosu and Feng Chen. Semantics and algorithms for parametric monitoring. *arXiv preprint arXiv:1112.5761*, 2011.

Verifying Good Practice Rules at Runtime with a Debugger

Raphaël JAKSE
Supervised by : Yliès Falcone, Kevin Pouget and Jean-François Méhaut
Laboratoire Informatique de Grenoble, UJF, CNRS, Inria, Corse

raphael.jakse@gmail.com, ylies.falcone@ujf-grenoble.fr,
Jean-Francois.Mehaut@imag.fr, kevin.pouget@imag.fr

Abstract

With the objective to help reducing the impact of software bugs, we define a way to ease discovery and understanding of a class of bugs related to good programming practices and good APIs or libraries usage. Our idea is to check properties relative to good practice and good usage at runtime using a debugger. In this paper, we extend the runtime property-checking model defined in a previous work and provide a efficient implementation of this extension. This extension is based on a trace slicing mechanism which allows efficient handling of parameterized properties, common among such good practice and good library usage properties.

1 Introduction

Writing softwares containing bugs is unfortunately common. A bug is a flaw preventing a system to behave as intended. By definition, eliminating bugs is desirable. Bugs can range from minor to critical. Minor bugs most likely cause annoyances while critical bugs can cost human lives.

One way of eliminating a software bug is to notice its existence by observing one of its consequences, find its cause in the source code and fix it. Fixing a bug is, in many cases, the easiest step of eliminating it when it is fully understood. Finding out its existence and fully understanding it, however, can both be really hard.

Bug discovery by manual testing. Several ways of discovering a bug exist. One of the most obvious ways is observing one of its consequences by using the software. Most obvious bugs can easily be spotted this way during the development of the software by testing manually modifications to the code or by a team responsible for testing the software Itkonen *et al.*. Bugs are also spotted by final users of the software, which, depending on the development model, the kind of software and the severity of the bug, is more or less undesirable.

Automatic testing. Automatic testing through unit tests can be used to ensure already fixed bug do not show again, to limit regressions and to check the code is correct for a restricted set of inputs Itkonen *et al.*. Research have been done on automatic generation of unit tests Cheon and Leavens Cohen *et al.*.

Preventing bugs with static analysis. Another common way of discovering bug is using static analysis or abstract interpretation Cousot and Cousot. With these methods, the source code of the software is analyzed without being run in order to find elements which are most likely programming errors or which might cause maintenance difficulties, raising the risk of introducing bugs during subsequent modifications. They can also prove some properties over the software's behavior. Unfortunately, while theses approaches are extremely valuable, they are not exempt of flaws: they can be really slow, limited to certain classes of bugs or properties, produce false positives and false negatives. While static analysis and abstract analysis can provide certain guaranties by proving properties over the program, proving correction of a software by static analysis is unsolvable in the general case Landi.

Theses approaches of finding bugs, as well as other approaches not covered in this introduction, have their own sets of drawbacks and benefits, making them all valuable for discovering different sets of bugs in different situations: some kind of bugs are more easily avoided by analyzing the software statically, other bugs are more easily and rapidly spotted by testing the software directly.

There are, however, bugs that are not discovered or understood easily or efficiently by aforementioned methods: as said above, static analysis cannot find every bug in all softwares and some bug are not apparent in an obvious way in all tests. When a malfunction shows during a test, its cause is not always easily understandable as the programming error at its root could be caused far before during the execution of the software (e.g. a pointer that should contain a valid address instead of a null or a random address).

Motivations. In this paper, we therefore present a technique complementing hereinabove methods, trying to gather their respective advantages, with a different set of benefits and drawbacks, thus providing a way of finding and understanding a class of bugs more efficiently and more easily in certain situations.

An entire class of bugs is caused by misuse of Applica-

tion Programming Interfaces (API) or libraries or by breaking common programming rules such as "do not modify a container while you are iterating on it, except under certain documented specific conditions". Such rules, as well as "good API/library usage", exist because if they are not followed, undefined or unattended behaviors can occur: while "iterating over a container in a given order" can be easily defined, "iterating over a muting container in a given order", under certain situations, can be hard, nay impossible to define, resulting in undefined behavior or runtime breakage. In the case of libraries or APIs, using them in a way that differs from the intended way can make the library user reach an untested and unforeseen case which can trigger bugs in the library or the application. The unintended use can also work, by chance, in the used version of the library, but the guarantee that it will work in a future version of the library is inexistent.

Goal. In this paper, we aim to present a method based on a debugger to verify, during the execution, "good practice" kind of rules, that is, rules inspired by common programming rules and properties describing correct API or library use.

We first introduce a simplified model inspired from the one we introduced in our previous work Jakse *et al.*. We then introduce an extension of this model to handle such properties efficiently. This extension is based on the concept of trace slicing as described and formalized by Roşu and Chen Rosu and Chen. We describe an efficient implementation of this extended model, used in our proof-of-concept.

2 Description of the Existing Property Checking Model

Debugger-based property checker. Properties described in this paper are checked over the execution traces of a software with the help of a debugger. The property checker is a whether a part of the debugger or an extension using a programming interface this debugger provides.

An execution trace is a sequence of events which can be monitored by a debugger, ordered by time. Properties are checked during the execution ("online"), meaning that while processing an event, the next event is not accessible. Theoretically, nothing prevents the trace from being infinite, though traces are always finite in practice and some properties might require the end of the trace to give a verdict.

Events which can be monitored by a debugger mainly include function calls. A function call event includes any value which can be accessed by the debugger before the first instruction of the function's body, including its parameters and the current value of global variables accessible from within the function. It also includes the return value of the function if, instead of considering the moment of the call, we consider the moment when the function returns. In practice, function calls are monitored through breakpoints. In order to monitor a function call, the property checker sets a breakpoint on the first instruction of the function's body and then asks the debugger for values needed to check the property. As an illustration for this model, see the architecture of our proof of concept in Figure 1, taken from our previous paper.

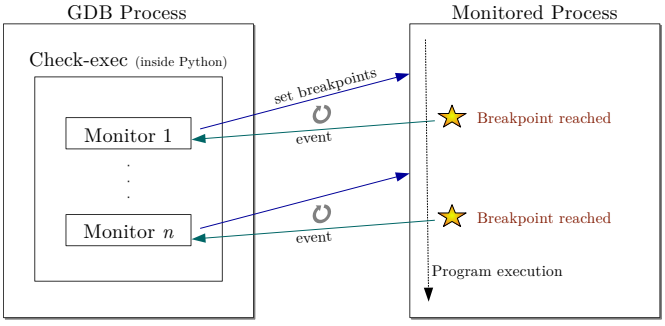


Figure 1: Architecture of `check-exec`, our proof-of-concept. GDB is the debugger on which we based our implementation. GDB provides a Python programming interface, we therefore chose to wrote our proof-of-concept in Python. A monitor is an instance of the property-checker. A monitor corresponds to one property.

Other events that can be monitored include syscalls and memory accesses (read, write or both). Syscalls can most likely be viewed as regular function calls in our model, though in practice monitoring syscalls involves different mechanisms such as catchpoints instead of breakpoints. Monitoring memory accesses, however, cause important performance issues in practice, thus needing more research in order to take advantage of this possibility. Both kind of events will be discussed in Section 4.1.

The advantages of basing the property checker on a debugger are discussed in our previous paper. They mainly consist in good performance, as the monitored program runs at native speed. Performance penalty are mostly caused by handling breakpoint and the time needed to change the property's state. Another advantage is that debuggers already are a familiar tool for developers so they do not need to learn a completely new tool to adopt this model. Checking properties at runtime, while limiting verification to tested executions, allows to handle more complex properties and do not require long code analyses, permitting to adopt the property checker in a fast modification-test loop based software development.

Automaton-based property description. In our previous paper, we defined a formalism to describe properties over execution traces. We designed this formalism with the idea that it should be efficient and feel as most natural as possible to developers, and be general or extensible enough to describe most properties. Therefore, we chosen to build a formalism inspired by state machines, which can be found in many computer science areas such as hardware design, human-computer interfaces and language theory. We here describe a simplified model which is sufficient for the needs of this paper but a more complex model is described in our previous paper, in which more details are given.

In this formalism, an automaton consists in a set of states Q which can be accepting or non accepting, a set of transitions δ, an initial state init and a initial variable environment σ_0 representing the initial memory of the automaton. A variable environment is a mapping from variable names to

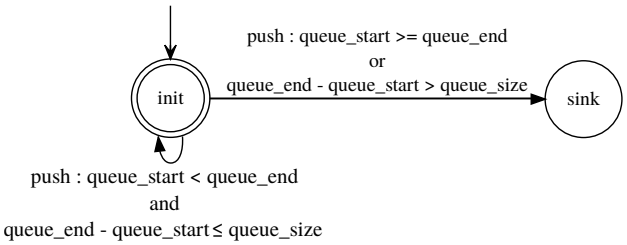


Figure 2: Example of a small property checking whether a queue overflows

variable values. The input symbols of the automaton are the events of an execution trace of the monitored program. See Figure 2 from our previous paper for a minimal example of an automaton in this formalism. In this example, the sink state is non-accepting while the init state is accepting. The init state is reached when a queue overflow occurs.

(q_k, σ_k) describes the internal state of the automaton before the k^{th} event of the trace, including its current state q_k and its current variable environment σ_k. The property described by the automaton is verified at the step k if and only if the current state is an accepting state.

A transition t consists in a start state $\text{start}(t)$, an event name, a guard $\text{guard}(t)$, an action and a destination state $\text{dest}(t)$. For each event concerned by the transition, the guard of the transition is a function depending on the values of the parameters values of the event and the current variable environment σ_k, returning a value in $\{\text{true}, \text{false}\}$. The guard decides if the transition must be taken. The action is also a function depending on the values of the parameters values of the event and the current variable environment σ_k, but contrary to the guard, it returns a new variable environment containing the same variable names as σ_k. At the beginning of the execution trace, the current state of the automaton is the initial state. For the k^{th} event e_k of the trace, transitions from the current state to a destination state describe the next internal state (q_{k+1}, σ_{k+1}) of the machine. For each transition t of the current state, if the event is concerned by the transition (the event name of the transition matches e_k), if the guard returns true, the new environment σ_{k+1} is given by the return value of the action and the new current state q_{k+1} is the destination state of the transition.

3 Extending the Model to Support Trace Slicing

3.1 Motivations

In this subsection, we briefly introduce trace slicing and we give examples of properties showing the need for trace slicing support in our model.

Many properties, including "good practice" and "good usage" properties, are to be checked on a per-object, or a per-family of object basis, rather than at a more global level. These parameterized properties are not to be checked against the entire execution trace. They are to be checked against each part (*slice*) of the execution trace specific to an instance of the object, or family of objects to which they relate. In the next paragraphs, we give examples of such parameterized properties.

Simple parameterized properties. The simplest parameterized properties are those involving one object.

First of such properties are those following the acquire-release resource model Rosu and Chen, in which each acquired resource must be released and no resource can be released without being acquired, represented bellow with an regular expression-like syntax:

$$\forall r, (\text{acquire}(r) \cdot \text{release}(r))^*$$

Or, in a negative way:

$$\nexists r, \text{begin} \cdot \text{release}(r) \,|\, \text{acquire}(r) \cdot \text{end}$$

The regular expression, in both cases, is not matched on the entire execution trace, but it is matched with every slice of the execution trace corresponding to an instance of the quantified parameter, that is, for each instance of r, the corresponding slice of the execution trace contains any event which does not involve any other instance of r.

Based on the same acquire-release model, any opened file should be closed:

$$\forall f, (f = \text{fopen}() \cdot \text{fclose}(f))^*$$

Reading from or writing to a file can only be done after it was open and before is was closed:

$$\forall f, (f = \text{fopen}() \cdot (fprintf(f)|fread(f))^* \cdot \text{fclose}(f))^*$$

Any allocated block memory should be freed:

$$\forall p, (p = \text{malloc}() \cdot \text{free}(p))^*$$

In C++, any created ("new"d) object should be deleted:

$$\forall obj, (p = new \cdot \text{delete}\, p)^*$$

Parameterized properties involving several parameters. There are also more complex properties involving a family of objects. This typically applies for object which are linked in one way or another. A good example of such family is collections and iterators. Meaningful iterators are always tied to a collection, and a corresponding rule is: an iterator must not be used after a modification of its corresponding collection if it was created before this modification. See Figure 3 for a description of this property with an automaton.

An example of slicing a fictive execution trace on this property is given in Table 1.

As there can be several iterators over the same collection, any event involving a collection without any particular iterator, like $\text{modify}(c)$, can concern several slices of the execution trace.

Slice	Parameter instances	Current state	Affected slices
Slice 1	(None, None)	init	
event: newCollection(c1)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(from slice 1)
event: newIter(c1, i10)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
Slice 3	(c1, i10)	iReady	(from slice 2)
event: next(i10)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	
Slice 3	(c1, i10)	iReady	(*)
event: newIter(c1, i11)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
Slice 3	(c1, i10)	iReady	
Slice 4	(c1, i11)	iReady	(from slice 2)
event: newCollection(c2)			
Slice 1	(None, None)	init	(*)
...	...	...	...
Slice 5	(c2, None)	cReady	(from slice 1)
event: modify(c1)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
Slice 3	(c1, i10)	invalidated	(*)
Slice 4	(c1, i11)	invalidated	(*)
Slice 5	(c2, None)	cReady	
event: newIter(c1, i12)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	(*)
...	...	...	...
Slice 6	(c1, i11)	iReady	(from slice 2)
event: newIter(c2, i20)			
Slice 1	(None, None)	init	
...	...	...	...
Slice 5	(c2, None)	cReady	(*)
Slice 6	(c1, i11)	iReady	
Slice 7	(c2, i20)	iReady	(from slice 5)
event: next(i10)			
Slice 1	(None, None)	init	
Slice 2	(c1, None)	cReady	
Slice 3	(c1, i10)	error	(*)
...	...	...	...
Slice 7	(c2, i20)	iReady	
event: next(i20)			
Slice 1	(None, None)	init	
...	...	...	...
Slice 7	(c2, i20)	iReady	(*)
event: modify(c2)			
Slice 1	(None, None)	init	
...	...	...	...
Slice 5	(c2, None)	cReady	(*)
Slice 6	(c1, i11)	iReady	
Slice 7	(c2, i20)	invalidated	(*)

Table 1: Trace slicing example with a property checking good usage of iterators and collections.

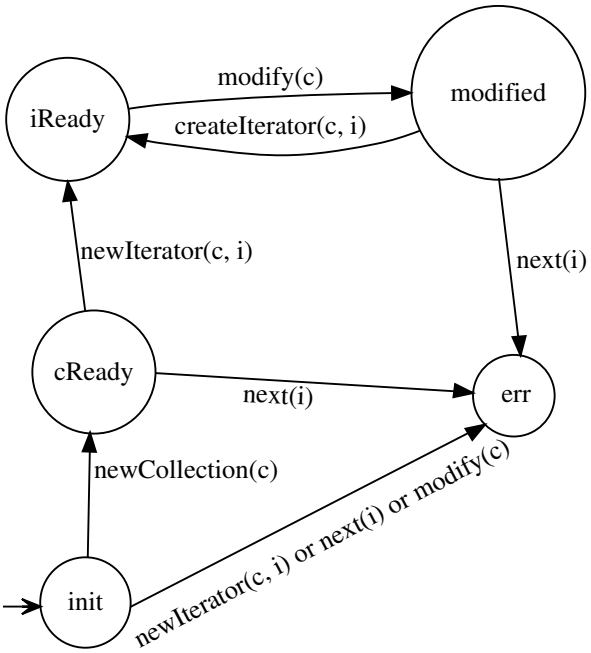


Figure 3: An iterator must not be used after a modification of its corresponding collection if it was created before this modification.

3.2 Slicing Automaton Model.

The existing model already handles parameterized properties like those presented in Section 3.1, being Turing-Complete thanks to the variable environment, the guard and the action functions of transition. However, it was designed for properties checking the state of a program at a global level. Writing parameterized properties in this model thus involves a lot of work that is duplicated for each such property and might even require "bypassing" the model by programming one in the automaton. The resulting automaton is thus not guaranteed to look natural and to be easy to understand. As such parameterized properties are common, we therefore define, in this subsection an extension to our model that allows to write parameterized properties efficiently and more naturally, while still handling unparameterized properties as a special case of parameterized properties with 0 parameters. Writing parameterized properties in this extension is similar to writing unparameterized properties in the base model, the only additional requirement being specifying the slicing parameters.

In the general case, though we know in advance how many parameters are involved in the trace slicing, as they are declared in the property with a mechanism similar to quantifiers, we do not know how many instances will be encountered while evaluating the property. We therefore need to be able to create slice dynamically during the evaluation.

Components of the slicing automaton. The slicing automaton model is defined by all elements of the base model, to which we add a tuple $S = (P_1, \ldots, P_m)$ containing the list of formal parameters on which the execution trace must be sliced.

$C_{k+1} \leftarrow \emptyset$
for all $s \in C_k$ **do**
  if $\exists t \in \delta, \text{start}(t) = q(s)$ and $\text{guard}(t)$ is verified **then**
    Let t be such transition
    $\sigma' \leftarrow \alpha_t(env(s), e_k)$
    if $\inf(I(e_k), I(s))$ **then**
      $s' \leftarrow (\text{dest}(t), I(s), \sigma')$
    else if $\text{comp}(C_k, I(e_k), I(s))$ **then**
      $C_{k+1} \leftarrow C_{k+1} \cup \{s\}$
      $s' \leftarrow (\text{dest}(t), \text{merge}(I(s), I(e_k)), \sigma')$
    else ▷ This slice is not concerned by the event
      $s' \leftarrow s$
    end if
  else ▷ This state is not concerned by the event
    $s' \leftarrow s$
  end if
  $C_{k+1} \leftarrow C_{k+1} \cup \{s'\}$
end for
[1]

Figure 4: Computing the next internal state of the slicing automaton

Before the event e_k, the current internal state C_k of the slicing automaton is a set of slices objects corresponding to the slices of the execution trace. Each slice object s contains a current state $q(s) \in Q$, a tuple $I(s)$ of parameter instances $(I(s)_1, \ldots, I(s)_m)$ and the variable environment $env(s)$. For any $i \in \{1, \ldots, m\}$, $I(s)_i = \text{None}$ if the parameter P_i of S is not instantiated in this slice.

The property described by the automaton is verified at the step k if and only if the current state of every slice of C_k is an accepting state.

Running the slicing automaton over a trace. At step $k = 0$, that is, before handling the first event, the internal current state C_0 of the slicing automaton is the singleton $\{s_0\}$ with s_0 such that $q(s_0) = \text{init}$, $env(s_0) = \sigma_0$ and $I(s_0) = (\text{None}, \ldots, \text{None})$.

That is, the initial internal current state consists in a single slice without any instantiated slicing parameter.

The internal state C_{k+1} of the slicing automaton at step $k+1$ is defined using the internal state C_k at step k and the k^{th} event, e_k, of the execution trace, according to the procedure 4.

We note:

- α_t the action function of the transition which, from a variable environment and the event, returns a new variable environment.
- $param(e)$ the function which maps a formal parameter of the event e to its instance in e.
- $I(e_k)$ the tuple of instances of the property's slicing parameters built from the parameter instances of the event e_k. If a slicing parameter instance is not given by the event, the corresponding component of the tuple is None.

The expression $\inf(i_1, i_2)$ checks whether each component of the parameter instance tuple i_1 is equal to $None$ or to the corresponding component of the parameter instance tuple i_2.

In the case i_1 is an instance tuple derived from an event and i_2 is an instance tuple of a slice, if $\inf(i_1, i_2)$ is true, we are in the case where the event belongs to the slice: every parameter instance given by the event is equal to the corresponding instance of the slice. If $\inf(i_1, i_2)$ is false, the event belongs to another existing slice or a slice which is to be created.

The expression $\text{comp}(C, I(e_k), I(s))$ checks whether e_k must lead to the creation of a new slice, given existing slices in C. This is the case if:

- $\forall i \in \{1, \ldots, \#(S)\}, I(s)_i \neq \text{None}$ and $I(e_k)_i \neq \text{None} \Rightarrow I(s)_i = I(e_k)_i$, that is, there are no incompatible parameters between parameter instances, and
- $\forall s' \in C, \inf(I(s), I(s'))$ is false or $I(e_k)$ and $I(s')$ are incompatible, that is, there is no slice at the same time more specific than s and compatible with the event parameter instances, "more suitable" for this event.

The expression $\text{merge}(I(s), I(e_k))$ gives a tuple of the same size of $I(s)$ and $I(e_k)$, in which the j^{th} component is equal to the j^{th} component of $I(s)$ if this component is not None, otherwise to the j^{th} component of $I(e_k)$, which can be None. If both components are not None, by construction of the procedure, they are equal. $\text{merge}(I(s), I(e_k))$ can be seen as a kind of least upper bound of $I(s)$ and $I(e_k)$.

4 Implementation of the Slicing Automaton Model

In this section, we describe an efficient implementation of the slicing automaton model defined in Subsection 3.2. A naive implementation would indeed be inefficient when dealing with execution trace containing a large number of instances of the slicing parameter tuple, as for each event of the trace, the set of slices would be entirely visited, and for each slice, this set can be visited a second time. Determining, at anytime, whether the property holds is also costly as it merely requires to loop over the set of slices to look for a non-accepting current state.

Our implementation, inspired by the algorithms described by Roşu and Chen Rosu and Chen and used in our proof-of-concept, make use of appropriate data "caching" structures to avoid looping over the set of slices altogether.

Optimizing access to the set of current states. In order to make this access efficient, we define `CSC`, a counter (multiset) which maps each $q \in Q$ to the number of slices of which q is the current state. Each time a slice is added, removed or updated, the `CSC` is updated. The multiset can be efficiently implemented as a hash table or, even better, by a simple array if states are represented by naturals. In both cases, obtaining the set of states means iterating over the set of states, listing each state q for which `CSC`[q] is not zero. If the set of state is expected to be large and the property often evaluated, a `curStates` array of $\#(Q)$ cells containing the list of current states and a `curStatesCount` natural giving the

number of current states can be maintained next to CSC, thus reducing the complexity of accessing the set of current states to a constant time.

Obtaining the set of related slices when an event is handled. The idea behind our implementation is to precompute the access of slices for future events when information needed to compute this access is easily accessible rather than when an event is handled. This information is easily accessible when we are creating new slices. At this time, it is possible to build the list of parameter instances tuples which are compatible with this slice and to connect these instances with this slice in a global map. These instances can contain wildcards for parameters which are not instantiated in this slice. When a event is handled, the parameter instance tuple is used to ask the global map to get the list of related slices.

We therefore define F, a table which maps a parameter instance tuple to a list of slices. The purpose of F is to be used as the global map described in the previous paragraph.

In Figure 5, we describe the procedure our implementation follows when an event is applied to a slice. $\mathrm{instance}(P, s)$ is the value of the formal parameter P in the slice s. If P is not instantiated in the slice $\mathrm{instance}(P, s) = \mathrm{None}$. Likewise, in In Figure 6, $\mathrm{instance}(P, e)$ is the value of the formal parameter P in the event e, or $None$ if e does not provide an instance of P.

The parameter compatSlices of the procedure APPLYEVENTSLICE contains slices which are compatible with the instances of the event's parameter instances. It is used in the expression $\mathrm{comp}(\mathrm{compatSlices}, I(e), I(s))$ in order to determine whether a more specific compatible slice exists. inf and comp have the same meaning as in the description of the slicing automaton model. However, in this implementation, we already built a list of compatible slices, so the implementation of comp does not need to check this again, only compatible slices are looped over, where a naive implementation could have made comp loop over the whole set of slices.

In Figure 6, we describe the procedure in charge of dispatching events to the right slices. The expression GETCOMPATIBLEF(p, s) builds, from the table F, the list of slices having instances p' which are less specific than p, where less specific means that each component of p' is whether equal to $None$, or to the corresponding component of p, that is to say, the slice s is added for all possible instance tuple that would be compatible with this slice.

In Figure 7, we give an implementation of GETCOMPATIBLEF which makes no assumptions on the internal implementation of the table F. Depending on the data structure used to represent F, this implementation can be optimized to reduce the number of operations. In particular, if F is implemented as a trie where components of tuples are used as its keys, prefix lookups can be gathered efficiently so the cost of inserting new tuple is reduced.

4.1 Limitations

Benchmark. The implementation described in Section 4 and the corresponding proof-of-concept still lack solid benchmarks to prove their efficiency. Unfortunately, we still need

```
1: procedure REGISTER(newQ, σ, i)
2:     create s
3:     dest(s) ← newQ
4:     env(s) ← σ
5:     I(s) ← i
6:     CSC[newQ] ← CSC[newQ] + 1)
7:     visitedEvents(s) ← eventId(newQ)
8:     let p be a tuple with #(S) components
9:     i ← 1
10:    for all P ∈ S in order do
11:        p[i] ← instance(P, s)
12:    end for
13:    ADD(F, p, s)
14: end procedure
15: procedure UPDATESLICE(s, newQ, σ)
16:    CSC[dest(s)] ← CSC[dest(s)] − 1
17:    CSC[newQ] ← CSC[newQ] + 1
18:    dest(s) ← newQ
19:    env(s) ← σ
20:    q(s) ← newQ
21: end procedure
22: procedure APPLYEVENTSLICE(s, e, compatSlices)
23:    if ∃t ∈ δ, start(t) = q(s) and guard(t) is verified
   then
24:        Let t be such transition
25:        σ' ← α_t(env(s), e)
26:        if inf(I(e), I(s)) then
27:            UPDATESLICE(s, (dest(t), σ'))
28:        else if comp(compatSlices, I(e), I(s)) then
29:            REGISTER (dest(t), merge(I(s), I(e)), σ')
30:        end if
31:    end if
32: end procedure
[1]
```

Figure 5: Computing the next internal state of the slicing automaton

```
1: procedure APPLYEVENT(e)
2:     let p be a tuple with #(S) components
3:     i ← 1
4:     for all P ∈ S in order do
5:         p[i] ← instance(P, e)
6:     end for
7:     compatSlices ← GETCOMPATIBLEF(F, p)
8:     for all s ∈ compatSlices do
9:         APPLYEVENTSLICE(s, e, compatSlices)
10:    end for
11: end procedure
[1]
```

Figure 6: Procedure to handle an event

```
1: procedure GETCOMPATIBLEFSTEP(p)
2:     if i < #(S) then
3:         res ← GETCOMPATIBLEFSTEP(p, s, i + 1)
4:         if i^th component of p is not None then
5:             p' ← p
6:             p'[i] ← None
7:             res ← res ∪ ADDCOMPATIBLEFSTEP(p', s, i)
8:         end if
9:     else
10:        res ← get(F, p) or default to ∅
11:    end if
12:    return res
13: end procedure
14: procedure GETCOMPATIBLEF(p)
15:    return GETCOMPATIBLEFSTEP(p, s, 0)
16: end procedure
[1]
```

Figure 7: Mapping a parameter instance tuple to all compatible slices

to build a decent set of good practice and good usage rules to be able to benchmark correctly. Moreover, an issue is affecting the performance of our proof-of-concept when setting a finish breakpoint, used to retrieve the return value of the function. Unfortunately, the return value often gives the addresses of objects to be checked, as well as failure codes which are also important elements. This issues needs to be worked out in order to measure performance correctly. However, we already have evidences that our optimizations are correct with our preliminary measures on toy properties[1].

Proofs. No proof has been written for algorithms given in Section 4 as of the writing of this paper. It is therefore unknown whether these algorithms are formally correct. However, based on previous proved work, any error in these algorithms should be recoverable.

5 Future Work

Monitoring memory. As mentioned in Section 2, monitoring access to memory in a debugger is likely to cause performance issues. Indeed, monitoring memory access is done using watchpoints. Watchpoints are efficient only if they are implemented by the hardware. Software watchpoints are slow, as they require checking whether monitored addresses in memory have been accesses after each assembly instruction. Unfortunately, in common CPU architectures, very few hardware watchpoints are available. In x86 architectures, a maximum of 4 hardware watchpoints can be set[2].

[1] we ran a test program opening and closing 1000 fake files along with a property-checker creating a slice for each such file. On a modest laptop, switching from a naive slice handling to a a slice handling using the optimizations described in the paper, we were able to divide the run time by nearly 10

[2] `https://sourceware.org/gdb/wiki/Internals%20Watchpoints`

This limitation reduces the set of objects which can be checked. For instance, C++ standard iterators are similar to pointers, making the property on iterators given in Section 3.1 more difficult to check. This property, however, is suitable for iterators provided by others libraries[3]. All the more, in languages such as C and C++, internal data of containers are often accessible to the developer, so it is possible to bypass containers' method, which can mislead a property checker which does not monitor memory. Further research is needed to address this issue.

Method and function overloading. The area of method and function overloading has not been explored yet in our model. This exploration is essential to make it useful in languages supporting overloading like C++.

6 Related Work

JavaMOP Chen and Roşu is the reference implementation by the authors of the trace slicing method used in the paper. As our implementation, it can be used to check (parameterized) properties at runtime. It applies to Java program and makes use of the monitoring features of the Java Virtual Machine, contrary to our model, which is based on a debugger and applies to a set of languages which is not restricted to the set of languages based on the JVM.

There is also a project aimed at checking good API usage. This project is SLAM and is restricted to system softwares, mainly drivers, on Windows. Unlike our work, it is based on static analyzing.

Valgrind is a framework to instrumentate binaries and check them for defects. It provides a way to detect memory related defects by a dynamic binary recompilation and instrumentation process of the sofware's machine code and by running it on a simulated CPU Nethercote and Seward. It provides a more comprehensive detection of memory related defects than our approach Nethercote and Seward, however our approach can be used to detect certain memory leaks more efficiently by writing a rule which checks that each manually allocated memory block is freed.

7 Conclusion

With this work, we introduce an extension to our existing model in the hope it will allow to build solid tools to help bug discovery and understanding relative to good programming practice. A solid set of properties taking advantage of this extension is yet to be built. We hope API designers and library writers will be willing to write rules for their products in order to help softwares using them be more reliable, with the additional benefits that these rules could be used as a complement to their documentation and usage examples.

References

[Chen and Roşu, 2007] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework.

[3] GLib, for instance, provide containers accessible through iterators. `https://developer.gnome.org/glib/stable/glib-Sequences.html`

In *ACM SIGPLAN Notices*, volume 42, pages 569–588. ACM, 2007.

[Cheon and Leavens, 2002] Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP 2002Object-Oriented Programming*, pages 231–255. Springer, 2002.

[Cohen *et al.*, 1996] David M Cohen, Siddhartha R Dalal, Jesse Parelius, and Gardner C Patton. The combinatorial design approach to automatic test generation. *IEEE software*, (5):83–88, 1996.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[Itkonen *et al.*, 2007] Juha Itkonen, Mika V Mäntylä, and Casper Lassenius. Defect detection efficiency: Test case based vs. exploratory testing. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 61–70. IEEE, 2007.

[Itkonen *et al.*, 2009] Juha Itkonen, Mika V Mantyla, and Casper Lassenius. How do testers do it? an exploratory study on manual testing practices. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 494–497. IEEE Computer Society, 2009.

[Jakse *et al.*, 2015] Raphaël Jakse, Yliès Falcone, Kevin Pouget, and Jean-François Méhaut. Verifying properties at runtime with a debugger. *(not published yet)*, 2015.

[Landi, 1992] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[Nethercote and Seward, 2007a] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

[Nethercote and Seward, 2007b] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[Rosu and Chen, 2011] Grigore Rosu and Feng Chen. Semantics and algorithms for parametric monitoring. *arXiv preprint arXiv:1112.5761*, 2011.

Routing OSGi Services Over Multiple Nodes

Luc LIBRALESSO

Supervised by: Bassem DEBBABI

1 Abstract

In a world where application resilience and implementation simplicity are key factors, service-oriented component model implementations have a major role.

This paper introduces Cohorte Routing. Cohorte Routing is an improvement of an *OSGi* implementation: Cohorte Framework. It improves the application connectivity. It also allows new communication ways like Serial or Bluetooth making it possible to use Cohorte framework on microchips.

For the first time, OSGi applications can run on embedded systems. *Cohorte Routing* makes the connection between two different worlds: OSGi and embedded systems on microchips.

Keywords

SOA, Remote services, routing service

2 Introduction

Increasingly, software needs to run for a long time and with minimal interruptions. In the case of the Internet of Things, failures can be very frequent. For instance, Bluetooth connection not available or a wire is unplugged. In this case, software have to continuously adapt itself to cope with environment changes. It can be an exhausting and repetitive task for the developer to handle all possible cases of failure.

Cohorte framework was designed to answer to this concerns. How to set up a dynamic and service-oriented component model that abstracts network layer. Cohorte routing was build in the aim to enhance Cohorte. It make a better use of connections between nodes. For instance, it makes it simple to design nodes on micro-controllers like Cohorte MicroNode. In some cases, it solves connectivity problems transparently. Cohorte Routing allows more features and improve the application connectivity.

The routing problem is about to create Path in a network with the objective to minimize each path between two nodes.

We have a set of peers V where each element $v \in V$

We also have a set of links between peers E where each element $e \in E$ is a couple $(v_1, v_2) \in V^2$. and a weight function $W : V^2 \to \mathbb{R}$.

Finally, we have a set of communication protocols for each peer $p : V \to P$ where $P \subseteq \{HTTP, XMPP, Bluetooth, Radio, \dots\}$ and $P \neq \emptyset$.

Example : In the case we have :

- $V = \{v_1, v_2, v_3\}$
- $E = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$
- W :
  - $W(v_1, v_2) = 1$
  - $W(v_2, v_3) = 3$
  - $W(v_3, v_1) = 2$
- $P = \{HTTP, XMPP, Bluetooth\}$
  - $p(v_1) = \{XMPP, HTTP\}$
  - $p(v_2) = \{HTTP, Bluetooth\}$
  - $p(v_3) = \{Bluetooth\}$

This instance is shown on the figure 1

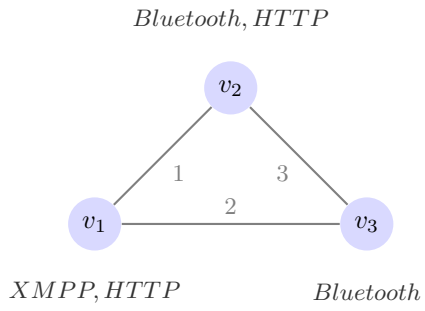


Figure 1: Minimal routing example

This example is very recurring. It happens a lot with *Cohorte Micronode*[1] because the *MicroNode* only have a Bluetooth connection and is paired with a peer with Bluetooth and HTTP connection. Without *Cohorte Routing*. The MicroNode cannot access to the application beyond it's neighbour. This is why Cohorte Routing have many applications.

[1] *Cohorte MicroNode* is a minimal version of the *Cohorte* framework on a microship like a *STM32*

It is also possible to simplify the problem :

As we can see, each peer can communicate with it's neighbours if and only if they have a common communication protocol. i.e. for two neighbours u and v, u and v can communicate if and only if $p(u) \cap p(v) \neq \emptyset$. This step is called *Protocol Elimination*.

In the minimal example, v_1 can dialog with v_2 because they have a common protocol, in this case $HTTP$. But, v_1 and v_3 cannot communicate because v_3 only have a Bluetooth communication and v_1 doesn't have a Bluetooth communication. Without a routing algorithm, v_1 can't communicate with v_3 at all, but with it, v_1 can send messages to v_3 by sending them to v_2.

This new representation allows us to simplify the problem representation and stick to classical routing problems.

Figure 2 shows the instance simplified with the protocol elimination.

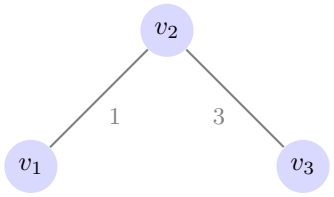


Figure 2: Minimal routing example after protocol elimination

The network topology can change rapidly. For instance, a new peer appears, a link between two nodes fall or a protocol on a node is temporarly unavailable. The algorithms should be able to cope with this variations and provide a new road in a short time.

The network can also be arbitrarly huge. So, the routing algorithm should be distributed.

The quality of the connections is also very variable. For instance, a Bluetooth connection between a *MicroNode* is tremendously slower than a HTTP connection between two nodes on the same computer[2].

Routing Algorithms Taxonomy : There are many routing algorithm families presented in [7]:

- *Distance Vector Algorithms* such as *RIP*[3]. Those algorithms generally use the *Bellman-Ford Algorithm*. It only knows it's direct neighbours at a first state. During the execution of the algorihtm, it will send information about it's knowledge of the network to it's neighbours. When a peer receives a routing information, it will compare it to it's own knowledge. If the received information is better than what it knows, it will keep it in their routing tables and propagate it to it's neighbours.

  If a node goes down, it's neighbours will notice that it is not responding and will assume it is down.

  With this kind of algorithms, each peer have partial knowledge of the network.

  Generally speaking, a routing table contains information such as :

  - Who are my neighbours ?
  - Which peer I know ? In this case, by which neighbour can I access it ?

- *Link-State Algorithms* such as *OSPF*[4]. By contrast with *Distance Vector Algorithms*, Link-State Algorithms keep each information about peers and links between them. Each peer has a global view of the network and can run a *Shortest path algorithm* like *Dijkstra algorithm* to specify the best next destination for the message.

Routing Metrics: All the routing algorithms choose a metric to base their computations of the best road. It mainly depends on underlaying goals of the routing.

Some examples of metrics :

- *Hop Count:* Used by *RIP*. The simpliest metric: it counts how many peers the message pass through to it's destination. It does not include the origin and the destination.
- *latency:* Time between the emission and the reception of the message.
- *bandwidth:* Data quantity for a fixed amount of time. It is generally used when someone have a lot of data to send.

3 Cohorte

Cohorte is a framework that allows to make dynamic distributed applications. It follows the OSGi specifications [1]. The developper is only interested in writting the component code. This code is automatically deployed by the framework. The developper do not have to be preoccupated by the network management and the fault detection which are repetitive and difficult task to cope with.

A Cohorte application is made by components that provides and consume services. Those components are executed in virtual machines called peers. And finally, those peers are executed in machines called nodes.

Cohorte framework is made of two parts presented in [2]:

- *iPOPO:* that implement the component system in different languages like *Python* or *Java*.
- *Herald:* that is designed to send messages from a component to an other. It abstracts all the network preoccupations in the component code. That is abstraction that allows a faster development of distributed applications and a better fault resilience.

[2] Almost a x100 factor

[3] Routing Information Protocol

[4] Open Shortest Path First

Example of iPOPO components: In this example, we will define two components: The fist providing a service and an other that uses this service. All the information needed by the framework is provided by *Python class and method decorators*.

Those decorators are explained in [6]

```
@ComponentFactory("ipopo-test-provides-factory")
@Provides('TEST_SERVICE')
@Instantiate('ipopo-test-provides')
class TestProvides:

        def __init__(self):
                [...]

        @Validate
        def validate(self, _):
                [...]

        @Invalidate
        def invalidate(self, _):
                [...]

        def method_provided(self):
                [...]
```

This Python code instantiate a simple component that provides a service called 'TEST_SERVICE'. It have :

- *Component Factory* manipulates the component class "ipopo-test-provides-factory"
- *Provides* tells iPOPO that this class of component provides a service called 'TEST_SERVICE'
- *Instantiate* tells iPOPO to instantiate the component as soon as it is possible
- a validate method that is called when the component is ready to start.
- a invalidate method that is called when the component is invalidate. i.e. when some requirements are missing.
- a method provided that can be called if an other component requires a 'TEST_SERVICE' service. This call is done by *Remote Procedure Call*.

iPOPO component that use the provided service:

```
@ComponentFactory("ipopo-test-requires-factory")
@Requires('_other', 'TEST_SERVICE')
@Instantiate('ipopo-test-requires')
class TestRequires:

        def __init__(self):
                [...]

        @Validate
        def validate(self, _):
                print('all requirements are ok')
                self._other.method_provided()

        @Invalidate
        def invalidate(self, _):
                print('Some requirements are missing')

        def some_method(self):
                [...]
                self._other.method_provided()
                [...]
```

In this component :

- the *Requires* decorator tells iPOPO that the component requires a service called 'TEST_SERVICE'. When the requirement is satisfied, the service will be accessible from the attribute *_other*.
- when *some_method* is called, it will call the method provided by the distant service 'TEST_SERVICE'. This method is similar to local code but executed on the distant host of the service provider.

The second component waits receiving a notification that says the first component have started. Then, the require is now satisfied and the component can be validated.

When the component is validated, the *validate* method is called and the fields required are injected directly in the component. So the programmer can now call other component method.

As we can see in this example, the programmer knows nothing about where the code is executed. The first component can be executed on a *STM32* in France and the other on a *supercomputer* in the USA.

More operations can be found in the *iPOPO documentation* like requiring optional services or aggregating a list of services accessible in the application.

4 Background & Motivation

Cohorte Routing was introduced in order to integrate *Cohorte MicroNode* and to improve an application connectivity. It was developped with in mind :

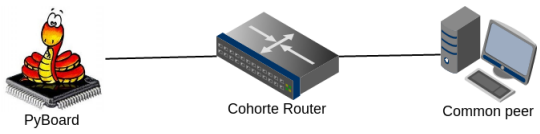


Figure 3: Cohorte Routing Use case : Cohorte MicroNode

Cohorte Routing main use case Cohorte MicroNode uses a bluetooth connection to interact with the application. But, it can be paired with one and only one peer. For accessing to other peers, *Cohorte Routing* is needed for allowing the router to send the microNode messages to the Common Peer. This topology is illustrated on figure 3.

- The application can be big (about 10^6 peers). With this constrint, the algorithm needs to be scaleable. Hence, *Link-State Algorithms* will have a complexity in memory of N^2. N is the number of peers for each peer. In our case, the algorithm will consume about 10^{12} Bytes, which corresponds to 1 teraByte of RAM and 10^6 teraBytes consumed for all the peers. It is too much, so we have chosen a *Distance Vector Algorithm* which has a complexity in memory of N. So, it will consume 1 megaByte for each peer and 1 teraByte in total which is much more honorable.
- The convergence time should be the shorter possible. It is important to detect promptly a connection loss. With this aim, the application needs a fast convergence.
- The changes must be retrocompatible. In the actual implementation of *Cohorte Herald*, there is no routing at all. It is important to preserve the correct execution of the application if some nodes have *Cohorte Herald* and some have not.
- The nodes need to be heterogenous. Some nodes like *Cohorte MicroNode* have a very few memory available. For some nodes less than 1 MB. With this constaint, we need to have different types of Nodes. Some of them cannot be routers. The implementation needs to be heterogenous and allow some nodes to consume less than 1 MB of RAM.
- Some links cost a lot and should be used in last resort. For instance, if we have a GSM[5] communication, the user pays for each Mega Byte of data sent. The application should be able to avoid the GSM usage if there are any other ways of communication.
- Because some peers have different computation capacity, some parameters like message frequency should vary from one peer to an other. Similarly, message frequency should change in the time. For instance when a congestion is detected on the network.
- The routing operations should be transparent for the application developper. The developper should not be concerned by the routing problem.

As we can see, there is a lot of constraints. That is why we need an other routing algorithm that can cope with all those needs.

5 Contribution

Cohorte Routing core algorithm is inspirated by *Babel algorithm* [3]. Due of different concerns, *Cohorte Routing* have a much simplier implementation that is component oriented. This implementation is presended below.

Algorithm 1 is executed at regular intervals defined by the HELLO_TIMER variable.

[5]Global System for Mobile communication

Algorithm 1 SendHellos : send *routing/hello* messages

```
1: function SENDHELLOS
2:     for v ∈ directNeighbours do
3:         if timer since last message elapsed then
4:             Send Routing/hello to v
5:             Memorize send date to calculate latency from
   v.
6:         end if
7:     end for
8:     for v ∈ peers that did not respond do
9:         if timer > HELLO_TIMEOUT then
10:            for i ∈ peers access is from v do
11:                metric[i] ← ∞
12:                nextHop[i] ← ⊥
13:            end for
14:        end if
15:    end for
16: end function
```

Algorithm 2 is executed at regular intervals defined by the variable ROADS_TIMER.

Algorithm 2 SendRoads : send messages of type *routing/roads* to each neighbour router

```
1: function SENDROADS
2:     for router ∈ neighbour routers do
3:         Send all information that is not provided by
   router
4:     end for
5: end function
```

Algorithm 3 is executed when a *routing/** message is received.

Algorithm 3 Handle : handle routing messages

```
1: function HANDLE
2:     if message is routing/hello then
3:         send routing/hello/reply to sender
4:     end if
5:     if message is routing/hello/reply then
6:         metric[sender] ← delta between now and send
   time of the hello message.
7:     end if
8:     if message is routing/roads then
9:         add changes from message in local routing table
10:    end if
11: end function
```

In figure 4, four components are used by *Cohorte Routing* :

Routing Handler: Is used to receive *hello messages* from a neighbour and respond to it.

Routing Hello: Is used to send *hello messages* to the neighbours to measure the metric in the link.

Routing Roads: Is used to send and receive information about roads. It gives access to a list of accessible peers through the network.

Routing Json: Is used to display information about routing table. It is useful for visualization of what is happening in a routing peer. It display a HTML table of the routing information. It also provides JSON format for the data for interaction with the Cohorte debug interface.

A *router node* must implement the four components and a *non-router node* only needs to implement *Routing Handler*

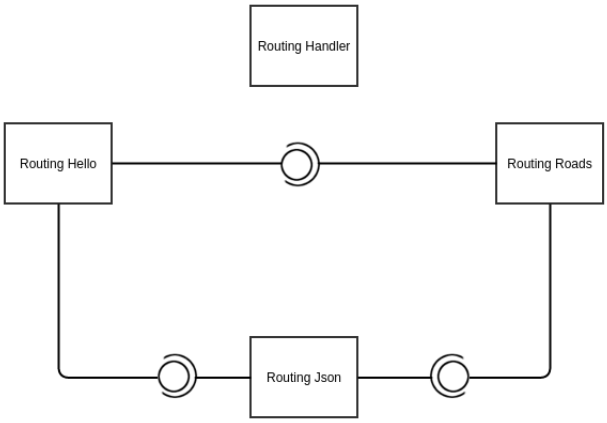


Figure 4: Component implementation of Cohorte Routing

Cohorte Routing Architecture: The Routing layer of *Herald* is inserted between the transport layer and the application layer. For retro-compatibility reasons, if there is a possible direct communication between two peers, the routing layer is not used. It is only used if necessary. The figure 5 shows how the routing layer interacts with Herald.

When a peer wants to send a message : If the sender has not the destination in its neighbours, it will use the routing layer. Depending on if the peer is router or none, two actions can occur :

- If the peer is router : The router will interogate the *Roads* component. If the peer exists, the message is sent to the next hop. It it not exists, an *Route Not Found* exception is raised.
- It the peer is not router : The routing layer will send the message to the last router that send a *hello message*. If there are no available roads, the router will notify it.

Wen a peer receives a message : It will check if it is the final destnation of the message.

- If it is, it will notify the application that a message is received.
- If it is not, it will try to send it to its final destination and similarly as previously, raise an error if the message could not be sent.

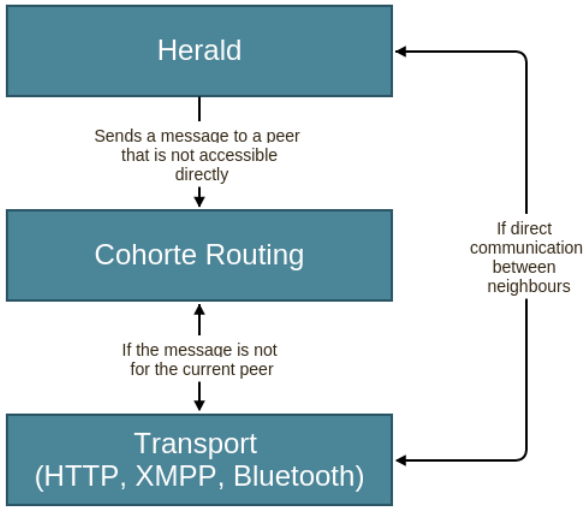


Figure 5: Cohorte Routing Implementation Architecture

6 Validation

In this part, we will validate the fact that *Cohorte Routing* answers to routing problem requirements.

- *Memory usage should be limited :*In *Cohorte Routing*, there is two types of peers :
  - *Router nodes:* keeps in memory for each of their neighbours, the metric information. And for each peer in the network, the next hop[6] and the total distance. The total complexity in memory is $O(N)$.
  - *Other nodes:* keeps only in memory the information about the last router that ask information. The total complexity in memory is $O(1)$.

  As said previously, if the application contains 10^6 nodes, in router nodes, the memory use will be about 1MB and not application size dependant on non routers. It is totally acceptable beacause the current memory capacity of computers now allows it and it is possible to make *microNodes* as non-routers.
- *Convergence Time should be short :* This point will be developped later.
- *Changes must be retrocompatibles :*With the *Cohorte Routing* architecture presented. The changes are totally retrocompatible with previous versions of the framework. In a heterogenous composition of nodes with a routing feature and nodes without routing feature, the routing nodes will keep in their routing table only nodes

[6]A next hop is the peer to contact for sending a message to a destnation. It is on the optimal path between origin and destination.

that are responding to their *hello messages*. In this case, nodes with an old version of Cohorte will continue to be able to communicate with their neighbourhood but will not be able to dialog with distant peers. Similarly, it would not be possible for a distant peer to acces to a old peer because it is not referenced in the routing table of routers.

- *Some nodes should have less work to do :* The subdivision in *router nodes* and *non router nodes* is helpful here too:
  - *router nodes:* will ask or send periodically for information in their neighbourhood. Moreover this period is perfectly customizable and heterogenous. In the same application at a same time, *router nodes* can have different parameters for the frequency. It will work perfectly fine.
  - *non router nodes:* only need to respond to *routing hello messages*.
  - *nodes with a previous version of cohorte:* can interact with nodes with a new version of cohorte if they are neighbours and should not have any additional work to do.

  With *Cohorte routing*, three different levels of work are available. Moreover, it is totally possible for an overloaded router node to become a non router node during the execution.
- *Some links should be used in last resort :* With the *metric measure* of *Cohorte Routing*, if a node has different links for accessing to a distant peer, it will select that which has the minimal metric. By setting links that should be used in last resort with a high metric, we guarantee that it will not be used if there is a better link for sending the messages. In the current version of *Cohorte Routing*,
- *Message frequency should vary during the execution :* From the fact that routing message sending is totally spontaneous, it is very easy to vary the sending frequency without disturb other peers. In the component oriented modelisation, the frequency factor is implemented as a property of routing components. By simply changing it, the frequency will automatically change.
- *The routing operations should be transparent :* The *Cohorte Routing* layer is set between the herald layer and the application layer. If a message is received, it is analysed and forwarded if the current peer is not the destination. And the destination of the message is checked and changed if needed by the routing. In this way, routing operations are transparent to the component developper and applications made with an old version of Cohorte will work fine on the new version.

Use Case: With the idea to test *Cohorte Routing implementation*, the following use case have been designed :

- On a *Cohorte MicroNode* was deployed serveral services to interact with electronical devices like LEDs or Temperature sensors.
- A gateway peer that is charged of redirecting messages from HTTP to Bluetooth.
- And finally, a peer with HTTP transport that have a component interacting with LEDs and sensors.

This use case is very common because it is the simpliest use of an electronical device interacting with a distant computer.

7 Related Work

There are many frameworks implementing service based component systems that allows components to provide remote services such as *eclipse remotes services* , *Apache CXF*, *Remote Operations Service Element (ROSE)*.

All of them provides ways to use *remote services* but do not provide a way to forward message through components. In some cases like using devices with low connectivity, it is a main feature.

There are also many routing algorithms.

- *RIP (Routing Information Protocol)* [4] : The *RIP* algorithm is a distance vector algorithm. It has a few memory usage but it does not take adventage of the metric. It uses the hop principle, so it would not privilegiate high bandwidths.
- *OSPF (Open Shortest Path First)* [5] : The *OSPF* algorithm is a link state algorithm. It keeps the network topology in order to execute a *Dijkstra algorithm* for the shortest path.
- *Babel algorithm* [3] : is a distance vector algorithm. It never suffers from routing loops and can use a metric such as the latency. *Cohorte Routing protocol* is inspirated by *babel* but it is simpler because it does not require sequence numbers for road information and *Cohorte routing* does not send requests for a new sequence number. It is a tradeoff that allows to exchange *convergence time* for *less trafic* from the routing algorithm. *Cohorte Routing* also allows a non-router node to receive messages from multiple router neighbours nodes without a memory complexity greater than $O(1)$.

8 Conclusion & Perspectives

This paper introduces *Cohorte Routing*. It allows *Cohorte* peers to forward messages from one peer to an other peer. With this comes a better application connectivity and the ability to weakly connected devices like *Cohorte MicroNode on STM32* to send message to the whole application through only one peer. This represents a way to port *OSGi* applications on microchips.

Moreover, *Cohorte Routing* was validated by a use case that allows distant components to interact with electronical devices through *Cohorte MicroNode*.

The choice for implementing *Cohorte Routing* with components is that it is easily extendable simply by instantiate new components that interacts with the routing components.

As one can see in the perspectives, many improvements can be implemented by a new component that is added in *Cohorte Routing* without changing existing code.

Perspectives:

- *links should be underrated :* In the current implementation of *Cohorte Routing* the link metric is fixed by latency. One would like to have a more customizable rating system for links. For instance, it is possible with a few modifications to create a component that provides a rate for different links. Its parameters can be the latency, the bandswidth, the connection type : i.e. GSM, Bluetooth, Ethernet, etc.
- *Performance Analysis of convergence time:* It would be interesting to compare *Cohorte routing* convergence time to other classical routing algorithms. The goal here is to evaluate the impact of message frequency in the global convergence time of the algorithm.
- A algorithm that changes the frequency of message sending. This is interesting because the convergence time should be augmented if many changes occurs in the network. Similarly, if there are a few changes in a large period of time, the algorithm can slow itself the number of routing message. It allows *Cohorte Routing* to adapt to its environement and reduce network traffic and convergence time when needed.
- Implementing a component that choose the default gateway for non-router nodes. With this system, if a non-router node have multiple router neighbours, it will change its default gateway only if it is unreachable.

References

[1] TO Alliance. Osgi service platform core specification, version 4.3, 2011.

[2] Thomas Calmant, Joao Claudio Americo, Olivier Gattaz, Didier Donsez, and Kiev Gama. A dynamic and service-oriented component model for python long-lived applications. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 35–40. ACM, 2012.

[3] Juliusz Chroboczek. The babel routing protocol. 2011.

[4] Gary Scott Malkin. Rip version 2. 1998.

[5] John Moy. Ospf version 2. 1997.

[6] Mark Pilgrim and Simon Willison. *Dive Into Python 3*, volume 2. Springer, 2009.

[7] Andrew S Tanenbaum. Computer networks, 4-th edition. *ed: Prentice Hall*, 2003.

Guide the Simulation to Achieve Functional Coverage

Lina Marsso
Argosim
Grenoble
marssolina@gmail.com.com

Supervised by: Bertrand Jeannet and Etienne Closse

August 21, 2015

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

Debugging a requirement allows detecting errors where there are written and not later when the effective implementation is tested against a requirement. Stimulus is a debugging requirement tools, which allows debugging requirement by simulating them. A simulation consists in generating random traces that satisfy the requirement.

In this article, we want to improve the traces generated by Stimulus by maximizing the functional coverage of a given requirement. To achieve this, we first clarify the notion of functional coverage, then we propose a heuristic to guide the simulation towards the coverage of the property. Finally we show the applicability of the proposed method thanks to the production of a prototype.

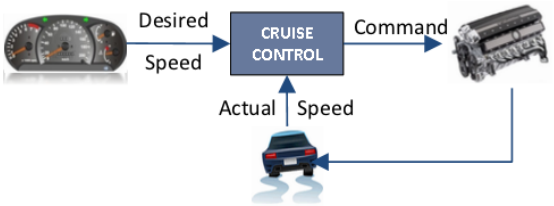
1 Introduction

Building a computer software is a complex task divided in different phases. The first ones consist in specifying the requirements and designing the architecture, the middle ones consists in effectively developing the software, and the last phases consist in validating the project by setting up and applying different test suites: unit, integration tests, ... In this paper we will focus on the first phase. This phase consists in deriving the requirements of the system from its informal specification. In this paper we focus on functional requirements of real-time systems.

Real-time systems are characterized by the fact that they continuously interacts with their environments. A typical example is a cruise control system of a car: this system periodically samples inputs from its environment, namely the desired speed required by the driver and the actual speed of the car, it performs computations and then issues commands to the motor. The functional requirements of such systems describe *what* they should do, but abstracts away *how* they should perform it. For instance, a requirement for a cruise control may be:

> *"When active, the cruise control shall not permit actual and desired speeds to differ by more than 3 kilometers/hour during more than 10 seconds."*

Observe that this requirement mixes a logical condition ("when active"), a numerical condition ("to differ by more than") and time ("during more than 10 seconds").



What are the actual, industrial practices regarding requirements? Most often, they are written in a natural language and validated by iterative reviews. This process can result in ambiguous or incorrect requirements. Furthermore, these problems are generally detected later in the test phases. The experience shows that the later the error is detected, the more expensive it is to fix the bug. Nowadays there exist tools to formalize and write the requirements of a project, like the B method [Hoang *et al.*, 2013]. One of them, Stimulus [Jeannet and Gaucher, 2015], enables in addition to simulate these requirements. We will focus more specifically on this last functionality.

Stimulus allows modeling real-time requirements with a formal yet close to natural language, in order to make models easy to write and to read. Its major feature however is to provide a simulation engine for generating execution traces that satisfy these requirements. Stimulus has its roots in the test modeling environments Lutin [Pascal *et al.*, 2008] and reactive programming languages Lustre [Halbwachs, 2005], Scade [Caspi *et al.*, 2003] and LucidSynchrone [Caspi *et al.*,]. The principle behind the simulation of requirements is to view requirements as constraints and to solve these constraints in order to generate execution traces which satisfies the requirements.

In this work we want to improve the traces generated by Stimulus. The idea is to guide the simulation and to select traces that maximize the *functional coverage* of a given requirement.

Functional coverage consists in covering a property. It differs from structural coverage which consists in ensuring test a piece of code has been activated during the execution of the program. Intuitively, in a simulation, a property is covered when it could have been violated. For instance, if the considered property is "$A \Rightarrow B$" and if in the simulation A is always false, the property cannot be violated, as it does not enforce any behaviour in such a context. Coming back to the cruise control example mentioned above, if we never have "active" then the requirement does not enforce anything, To really cover the property, a simulation should produce two speeds differing by more than 3 km/h, which gives the opportunity to check that such a situation does not last more than 10 seconds. In this example, interesting traces are traces which reach some covered configurations, in which the property is effectively tested.

The first objective of this paper is to clarify the notion of functional coverage of a property in the context of Stimulus. Ideally, we would like to automatically equip any property with an oracle indicating whether such an covered configuration has been reached during the execution.

Now, there are two options to effectively obtain execution traces which covers a given property: either one manually defines a specific scenario which will guide the simulation, or we design a method which automatically guides the simulation towards the coverage of the property. The second option is of course preferable as it is less demanding for the user. Such a guiding feature has already been investigated in black-box testing of reactive systems, for instance in the TGV [Jard and Jéron, 2005], STG [Ployette *et al.*,] and Gatel [Marre and Arnould, 2000] tools. Its use in the Ludic debugger for the Lustre synchronous language has also been investigated in [Gaucher *et al.*, 2003]. The approach proposed in [Ployette *et al.*,],[Gaucher *et al.*, 2003] to achieve this goal basically consists of

1. computing the set S of interesting configurations that can lead to one covered configuration (the ones in which the coverage oracle becomes true);
2. exploiting the knowledge about S to maintain the simulation as long as possible in S and to stop it if for some reason one do not succeed to maintain it within S;
3. at last, effectively guiding the simulation towards an covered configuration with some heuristics.

The second objective of the paper is to adapt the approach developed in [Gaucher *et al.*, 2003] to the context of Stimulus.

The rest of the paper is organized in the following way: we first present in section 2 an example of real-time functional requirement and how it is formalized with Stimulus. In section 3 we define a notion of functional coverage in the context of Stimulus. Then we explain in section 4 the approach we want to follow in order to achieve functional coverage of a property in a simulation. We terminate with a description of our realisations and experiments.

2 Stimulus on an example

In this section we use an example of automatic headlight requirements to describe the simulation process of Stimulus. .

If the switch is in the AUTO state then the headlights turn on or off, depending on the ambient light intensity, with a defined hysteresis to prevent blinking.

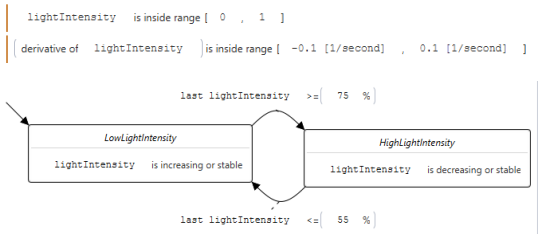
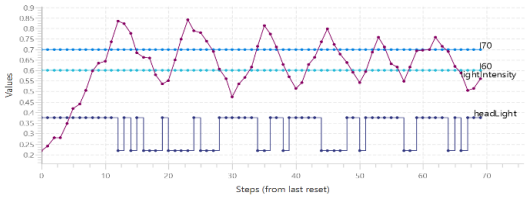


Figure 2: Environment assumptions

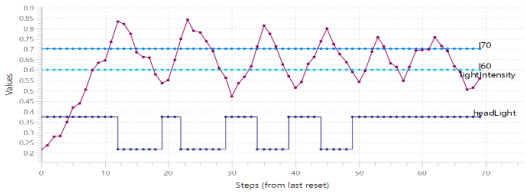
- *REQ_003Aa: if the switch is turned to AUTO, and the light intensity is at or below 70% then the headlights should stay or turn immediately ON. Afterwards the headlights should continue to stay ON in AUTO as long as the light intensity is not above 70%.*
- *REQ_003Ab: if the switch is turned to AUTO and the light intensity is 70%, then the headlights should stay or turn immediately OFF. Afterwards the headlights should continue to stay OFF in AUTO as long as the light intensity is not below 60%.*

These two requirements are formalized by using the library of predefined sentence templates provided by Stimulus. The resulting Stimulus model is depicted in figure 1. These two requirements use four sentence templates: *When, DoAfterwards, Initially* and *If*. Their definition will be described in the next section.

Simple assumptions on the environment of the system are formalized and shown in figure 2. The assumptions can be part of the requirements or are used to improve readability of the simulation results. They are a mix of standard patterns (domain range, stability property) and user-defined properties. In case of these two requirements, we are interested in testing the crossing of the *"lightIntensity"* barrier.



(a) sequential behaviour



(b) cyclic behaviour

Figure 3: Graph generate

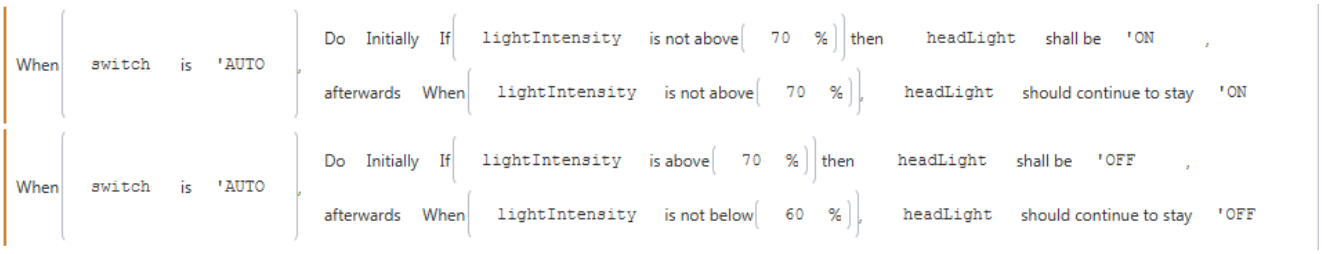


Figure 1: Two requirements formalised with Stimulus

Finally we generate traces, shown in the graph (a) in the figure 3, and observe manually if they satisfy the requirements. We observe in the traces this graph that after the headlights have been set to "OFF" the first time, the headlight behavior becomes random. Thanks to the result of the simulation, we know the existence of a problem, now we have to fix it. Effectively the requirement *REQ_003Aa* is ambiguous because we use the operators "as long as", that can mean *"as long as condition, something [afterwards nothing]"*, a sequential behaviour or *"[always] when condition, something"*, a cyclic behaviour. Graph (b) in the figure 3 confirms that here the second definition of as long as in the requirement *REQ_003Aa* have to be used.

3 Clarify functional coverage

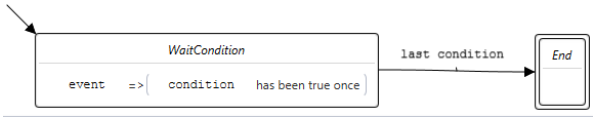
Requirements are edited and formalized thanks to the library of macros provided with Stimulus. We first give an intuition for each of these macros of what is the relevant coverage criteria. We then infer a general criteria which can apply to all library macros and user macros.

3.1 Simple macros

In order to propose a criteria, we need to introduce some notion. Each macro is associated with an automaton A. The notion of termination of an automaton will be exploited. Basically, an automaton A is terminated when the control is in a sink state (this means that there is no outgoing transition from this state) and the content of this sink state is itself terminated. This notion of termination will be used to decide whether a property specified with a macro is covered or not.

Now we present the most common macros used for specifying properties and we will discuss for each of them the relevant coverage criteria that applies.

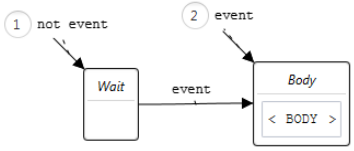
OnceBefore is associated with the sentence template "Ensure ⟨condition⟩ once before ⟨event⟩" and the macro is defined by the automaton



This sentence template enforces the behaviour: "the condition must be true at least once before or at the first occurrence of the event". An example of use is: "Ensure PlaneIsReady once before TakeOff". In this automaton, the "**last** condition" refers to the value of "condition" at the previous execution step. The automaton, hence the sentence template, terminates the step after the event condition occurs.

It is clear that to decide whether such a property is satisfied or not, one has to wait for the occurrence of the "condition", which is implied by the termination of the automaton. Hence the relevant coverage criteria consists in saying that the property is covered when its corresponding automaton terminates.

From is associated with the sentence template "From ⟨condition⟩, do ⟨BODY⟩" and is defined by the automaton



This sentence template enforces the behaviour: as soon as "condition" is true, enforce the behaviour of "BODY". An example of use which also exploits the previous template is: *From alarm, do Ensure acknowledge once before not alarm*.

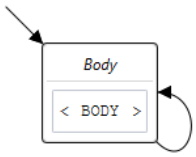
To decide whether this example property is covered, one has to wait for the occurrence of "alarm", and from there for the first occurrence of "not alarm". This is implied by the termination of the automaton associated with **From**, which occurs when the control in the state "Body" and its content "BODY" (here another sentence) is terminated.

At first glance the criteria for this sentence is the same as the criteria found for the previous one: the property is covered when the associated automaton terminates. However, the content of "BODY" might specify a cyclic, non-terminating behaviour, as we will see later.

Hence, our definition of coverage here will rather be: "From ⟨condition⟩, do ⟨BODY⟩" is covered when "BODY" is covered.

3.2 Cyclic macros

Repeat is associated with the sentence template "Repeat ⟨BODY⟩" and is defined by the cyclic automaton

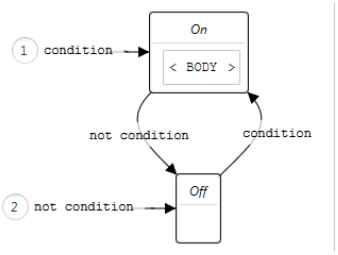


This sentence template allows to repeat a sentence once it is terminated. An example of use which also exploits the previous template is: *Repeat (From alarm, do Ensure acknowledge once before not alarm)*. To decide whether this example property is covered, one has to wait when the control in the state "Body" and its content ⟨Body⟩ is terminated once.

Hence, our definition of coverage here will rather be: "Repeat ⟨BODY⟩" is covered when "BODY" is covered once.

If we cover "Repeat" when we cover the body once, then there are no differences between in body inside the "Repeat" and the body alone. With other words if we cover only one time the body, we can not check the fact that this pattern is cyclic. The second idea is to cover the body. Covering the body twice and not thrice or N times is a choice totally arbitrary. That is one of our opened questions: Should coverage imply covering ⟨BODY⟩ once or N times ?

When is associated with the sentence template "When condition, ⟨BODY⟩" and is defined by the cyclic automaton

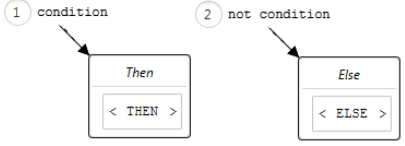


This sentence ensures a certain statement. The associated automaton has two transitions to ensure the condition. For instance the following requirement: *When alert (From alarm, do Ensure acknowledge once before not alarm)*. To decide whether this example property is covered one has to wait for the occurrence of "alert", and from there wait like in the previous macro "Repeat" the control in the state "Body" and its content ⟨Body⟩ are terminated twice.

Hence, our definition of coverage here is the same that the previous one ("Repeat").

3.3 Specifics cases

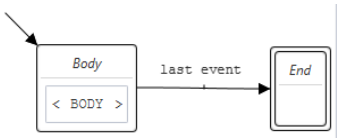
IfThenElse is associated with the sentence template "If condition then ⟨THEN⟩ else ⟨ELSE]⟩ and is defined by the automaton



This sentence template allows to choose a statement between two statements according to a condition. An example of use is: *if alert then damage = 0 else damage = damage + 1*

In this case we have to reach two final states if we want to cover all the properties, i.e. to terminate one time with damage = 0 when alert is fired and in a second time to terminate with damage = damage + 1 when alert is not fired.

Until is associated with the sentence template "⟨Body⟩ until event " and is defined by the automaton



This sentence template ensures a statement until an event happens. It uses the notion of a previous value of the variable that we saw with the operator "OnceBefore". An example of use is " (When alert (From alarm, do Ensure acknowledge)) until not alarm"

To decide whether this example property is covered one has to wait for the occurrence of "alert", from there wait until the content ⟨BODY⟩ is terminated and then wait for the occurrence of ⟨event⟩ (here not alarm).

Hence, our definition of coverage here will rather be: "⟨Body⟩ until event " is covered when ⟨BODY⟩ is covered and then ⟨event⟩ occurred.

The common principles of comparative template definition of coverage is to give and maintain values to cover all properties and the entire body. For the acyclic pattern, covering implies termination. We don't know if cyclic patterns coverage imply covering ⟨BODY⟩ once or N times.

For the sentence templates provided by Stimulus, we gave an definition. We didn't give one general intuition that works for all patterns. The future users have to give definitions for their added patterns.

To implement template definitions of coverage in stimulus is currently not possible. The main issue is that it is not possible to know if we cover the whole ⟨BODY⟩ in Stimulus. The idea is to complete the return value "True" with the last trace generated.

4 Guiding the test to achieve functional coverage

After defining the functional coverage, we adapt the approach developed in [Gaucher *et al.*, 2003] to the context of Stimulus to design a method which automatically guides the simulation towards the coverage of the property, in this section we describe this approach. It is divided into three steps:

1. computing the set $S_{interesting}$ of interesting configurations, i.e. the set of a configuration allowing to reach a final configuration;
2. maintaining the simulation if and only if we can reach an interesting configuration; and

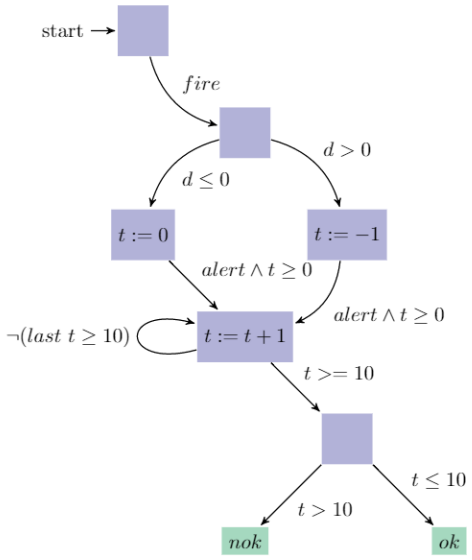


Figure 4: Example A: Flat automaton

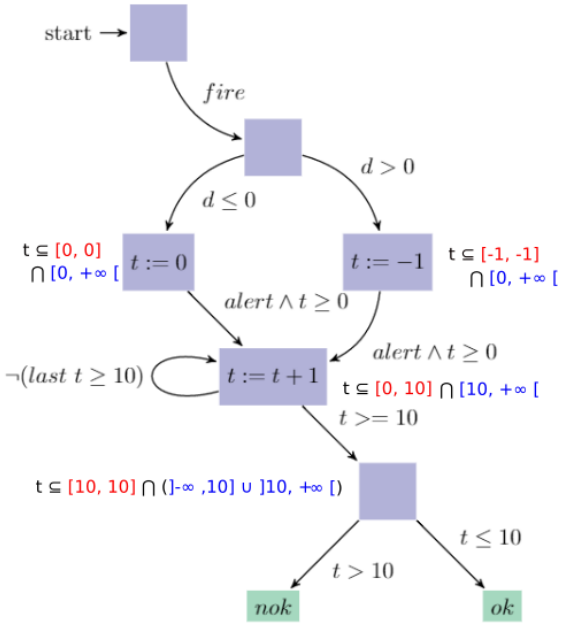


Figure 5: Flat automaton with for each state a set of approximation variables values, in red by forward analysis and in blue it is by back analysis.

3. guiding the simulation towards an covered configuration.

We illustrate this approach with the following requirement example A.

From fire,
Do (if $d \leq 0$ *then* $t := 0$ *else* $t := -1$*))*
Afterwards
Do ((from $(alert \wedge t \geq 0)$*,* $t := t + 1$*),*
until $t >= 10$*)*
Afterwards (from $t \leq 10$*, ensure acknowledgement)*

We represent the causal chain corresponding of the requirement example in the flat automaton in the figure 4. The goal of this approach is to reach one cover state (green states).

4.1 Computing the set $S_{interesting}$ of interesting configurations that can reach a coverage criteria

In order to define this approach, we need to introduce a preliminary notion. Each configuration is a control state s in an automaton A and the set of possible values for all variables at this current control state s.

The goal is to find necessary condition to stay in an interesting configuration. To compute an exact the set $S_{interesting}$ of configurations, we need to manipulate sets of Boolean and numerical relations. However numerical values are infinite and computing an exact set is an undecidable problem. We need to approximate these sets, that is why we use abstract interpretations to compute the sets of configurations.

To find an interesting configuration we need a co-reachable analysis and an reachable analysis. A co-reachable analysis gives the successor state s and a reachable analysis gives a predecessor of a state s. For each state s we compute the approximation of possible valuations for all variables thanks to a abstract interpretation tool. An instance of result of such an analysis is illustrated in figure 5.

4.2 Maintaining the simulation in $S_{interesting}$

The simulation have to continue if and only if we can reach an interesting state $s_{interesting}$ to give the chance to terminate in a covered state $s_{covered}$. A path could end up in a sink and in this case we can never terminate. For instance in the last example (figure 5), if we have fire and damages, such as fire = true and $d > 0$, we reach a sink. During the simulation if we reach a state which leads to a sink then we stop this simulation. However during the simulation we can still encounter a cycle, that is why need to guide the simulation towards a covered configuration.

4.3 Guiding the simulation towards a covered configuration

In order to define our approach to guide a simulation, we need to introduce two notions. The distance $\delta(s)$ for a state s is the minimum number of intermediary states before reaching one of the covering states and the weight is the likelihood to go from one state to another. Our approach combines these two notions.

The approach to guide the simulation, increases our chances to reach a state that can lead to a covering state. Concretely we associate a likelihood to each state s. The larger the chances are for a state to lead to interesting state, the greater the likelihood will be (figure 6).

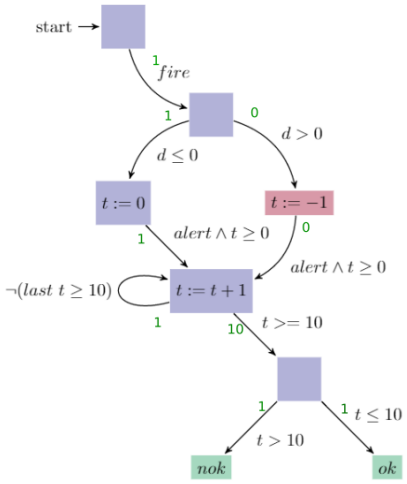


Figure 6: Flat automaton with likelihoods for each edges added in green

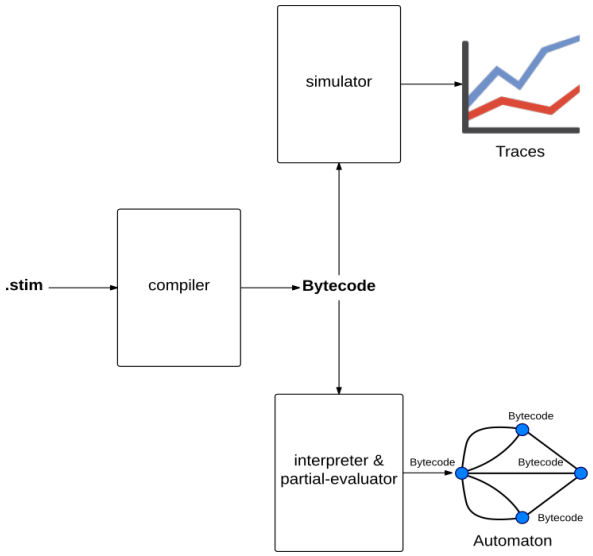


Figure 7: The functioning of Stimulus

5 Realisation and experiments

After defining the approach to guide the simulation, we describe the corresponding implementation in this section. It is divided into three steps:

1. generating automatically a flat automaton corresponding of programs written in Stimulus;
2. optimizing the automatons generated;
3. applying the analysis tools on this automaton.

The generation of flat automaton took more time than we thought because of the complexity of the language. The prototype is yet to be finished, therefore we only present a preliminary experimental report.

5.1 Generation of flat automaton

In order to explain how we generate the flat automaton, we need to explain how the Stimulus compiler is working thanks to the schema in figure 7. The compiler of Stimulus compile the program (.stim) to a Bytecode file. Then the simulator generates traces by interpreting the Bytecode and using a solver.

Because Stimulus is a complex language, we decided to construct the automaton by interpreting the Bytecode produced by its compiler instead of starting from relatively scratch. An interesting consequence is that our interpreter is independent from the semantics of Stimulus. It depends only on the semantics of the Bytecode. Also note that the interpreter performs a partial evaluation since it ignores complicated cases, for instance the notion of termination or the nested automatons.

As presented in section 3, each macro in Stimulus corresponds to an automaton A. For instance, the automaton corresponding to the macro "When", has one state "ON" and one state "OFF". Each automaton A has its corresponding variable called clock in the synchronous world by M. Pouzet in [Colaço and Pouzet, 2003]. Each states s_a in the automaton A correspond to a clock value ck_{value}.

We write a new interpreter and partial evaluator in order to generate the flat automaton. First, the interpreter computes the initial clock. Then the successor clocks, constraint labels and assignments are recursively computed by interpretation and partial evaluation of the Bytecode. After interpreting and partial evaluating we build the automaton such as:

- each state s is identified by a set of clock values ck_{value};
- each state s contains a list of statements;
- each edge e is a list of constraints and reset values.

For instance, the automaton shown in figure 8 is the result of the interpretation of the requirement *From alarm, do Ensure acknowledge once before not alarm*. In this automaton, the vertices contain values of clock variables ck and the assignments whereas the edges contains the constraints and values of reset variables r.

5.2 Optimizations

The analysis of an automaton has a cost proportional to the number of edges and vertices of the automaton. To prevent this cost from becoming too large, we optimize the automaton before beginning the analysis. All our optimizations are performed directly when building the automaton. The most important optimizations are the following:

- Some vertices can be deleted when one constraint c_1 is included in another constraint c_2, such as $x \geq 5$ and $x \geq 10$. For instance the automaton in figure9 written with Stimulus and its corresponding flat automaton optimised (figure 10).
- Unreachable edges, i.e. edges with contradicting constraints, can be deleted. For instance, $\neg x \wedge x$.
- The values of the resets are propagated forward, allowing to simplify the contents of the state, and to simplify the expressions processed during the analysis

The previous optimisation have certain limitations. For instance, the optimisation such as one condition $c1$ is included in a other condition $c2$ do not work in the case where we have a time value. Indeed, in Stimulus we do not know the equality

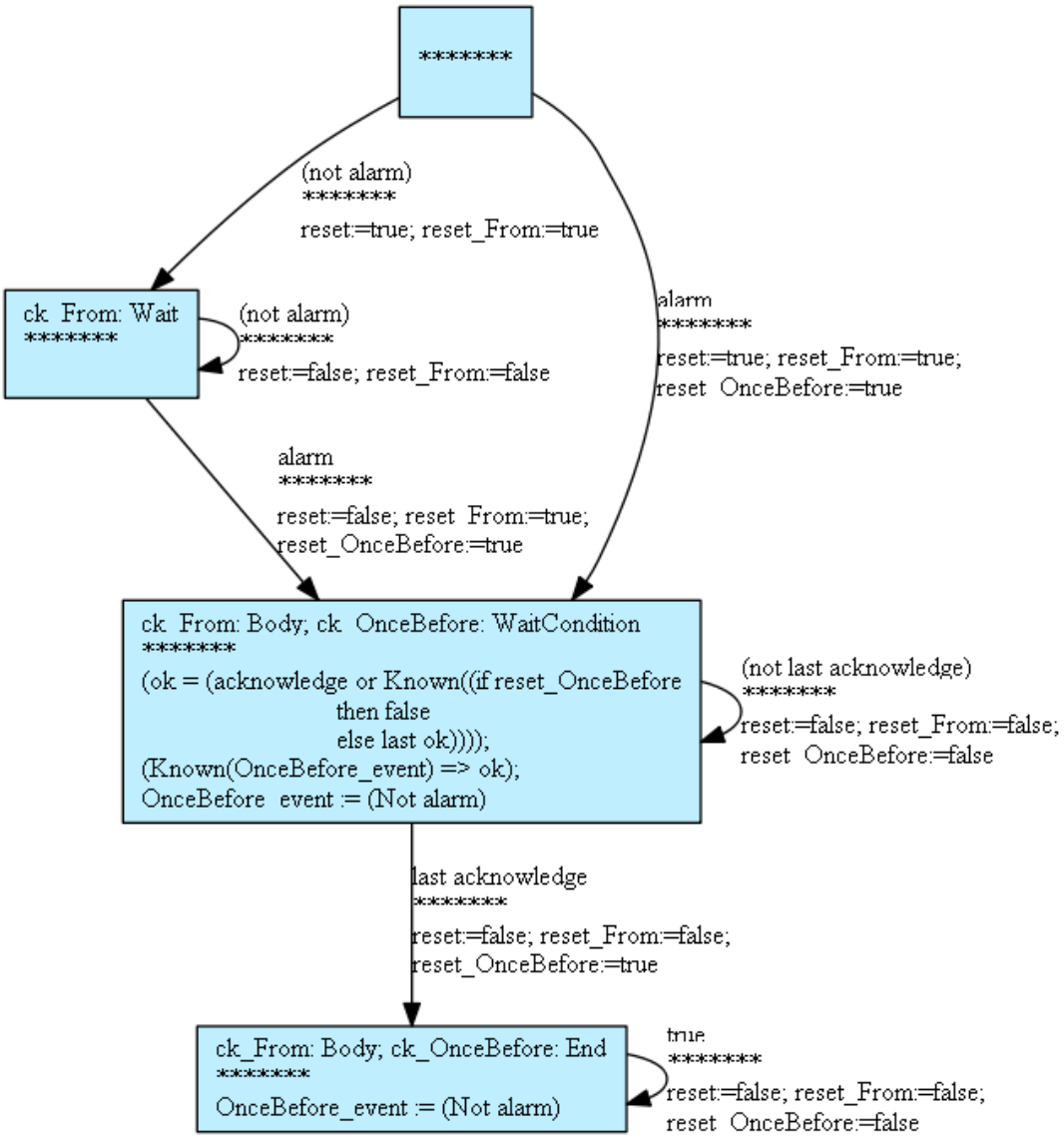


Figure 8: Exemple of flat automaton generated

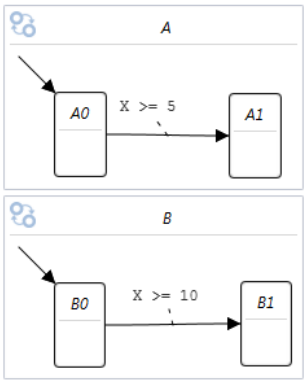


Figure 9: Two automatons in parallel using input variables

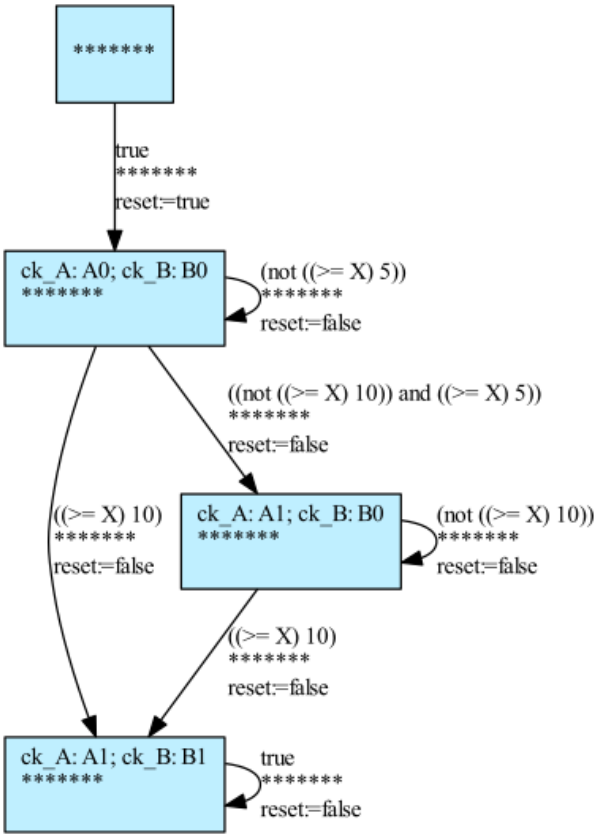


Figure 10: Flat automaton generated corresponding to the automaton in figure 9

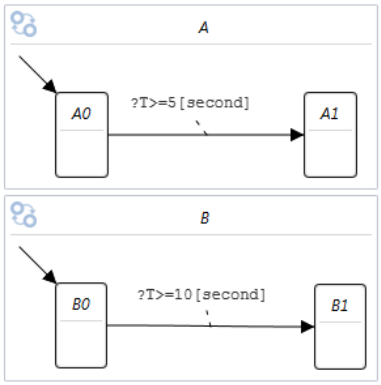


Figure 11: Two automatons in parallel using time variables

of the time in two different automatons in parallel. We illustrate this case in the two following automatons. The first one is the automaton written in Stimulus in the figure 11, and the second one is the automaton generated in the figure 12.

We observe that the number of vertices and edges is larger than in last example (figure 9). The only differences between the two examples is that the constraints concern the time and not a simple variable.

5.3 Experiments

Our interpreter is able to automatically generate an automaton corresponding to a program written in Stimulus. They are only two unsupported structures: arrays and iterators from Stimulus.

Since Stimulus is an industrial tool, its language is quite complex and implementing the generation of automaton was tedious.

We did not tackle the automation the guiding of the simulation towards a configuration. However, it is already possible to apply certain analysis tool on the automaton, provided that the tool supports Boolean and numerical variables.

Today, we can guide the simulation in the automaton by adding likelihoods manually on ours automatons.

6 Conclusion and Further work

Debugging requirements by simulation is the goal of Stimulus. Simulating a requirement consists in generating random traces that satisfy the requirement. In this paper we presented how to improve the traces generated by Stimulus. Our approach consists in guiding the simulations towards a coverage goal.

For this purpose, we first clarified the functional coverage by giving descriptions of how to cover the sentence templates provided by Stimulus. This clarification highlighted a lack in Stimulus. It is not possible to know what we exactly cover in Stimulus. We need to add more information in the return value of a simulation. The solution will be to allow to give an argument ⟨BODY⟩ of a macro in the language Stimulus.

Secondly, we described precisely our methods to guide the simulation towards the coverage of the property thanks to the approach developed in [Gaucher *et al.*, 2003] .

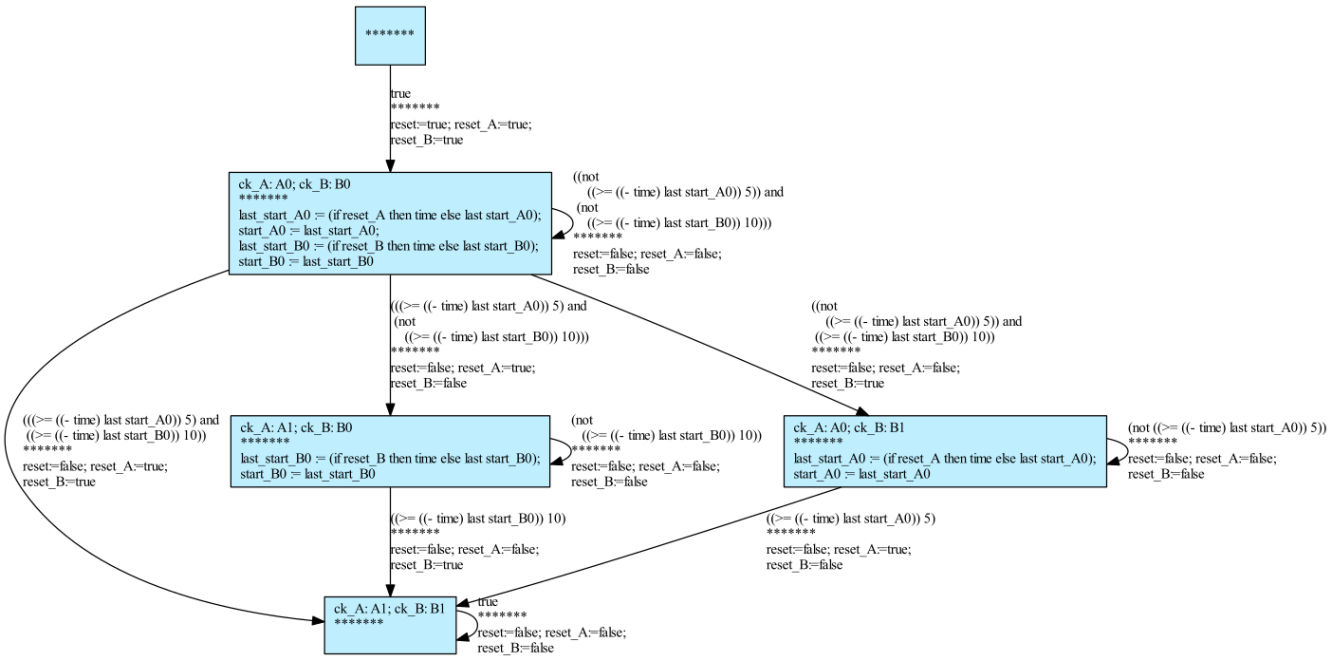


Figure 12: Flat automaton generated corresponding to the automaton in figure 11

Finally we began the implementation of these methods in Stimulus. Concretely we implemented an interpreter to generate a flat automaton of the system. Working at the Bytecode level allows to be independent from the semantic of Stimulus.

In its current state, we can apply existing analysis tools on our automaton as long as the tool supports Boolean and numerical variables.

We identify two main directions to explore as future work:

- apply more complete analysis technique on the automatons generated;
- automate the detection of cover states and the delivery of likelihoods

Acknowledgments

First of all, I thank Bertrand Jeannet, my internship supervisor and co-founder of ARGOSIM. I am extremely grateful and indebted to him for his expertise, sincere and valuable guidance and encouragement extended to me. I take this opportunity to record my sincere thanks to Etienne Closse, my second supervior and also co-founder of ARGOSIM and all other members of Argosim for their help and encouragement. Finally, I place on record, my sense of gratitude to one and all who, directly or indirectly, have lent their helping hand in this project.

References

[Caspi *et al.*,] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Lucid synchrone, un langage de programmation des systèmes réactifs. *Systèmes Temps-réel: Techniques de Description et de Vérification-Théorie et Outils*, pages 217–260.

[Caspi *et al.*, 2003] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, volume 38, pages 153–162. ACM, 2003.

[Colaço and Pouzet, 2003] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Embedded Software*, pages 134–155. Springer, 2003.

[Gaucher *et al.*, 2003] Fabien Gaucher, Erwan Jahier, Florence Maraninchi, and Bertrand Jeannet. Automatic state reaching for debugging reactive programs. In *the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, 2003.

[Halbwachs, 2005] Nicolas Halbwachs. A synchronous language at work: the story of lustre. In *Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.*, pages 3–11, 2005.

[Hoang *et al.*, 2013] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-b patterns and their tool support. *Software & Systems Modeling*, 12(2):229–244, 2013.

[Jard and Jéron, 2005] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[Jeannet and Gaucher, 2015] Betrand Jeannet and Fabien Gaucher. Debugging real-time systems requirements : Simulate the "what" before the "how". *Embedded-World*, 2015.

[Marre and Arnould, 2000] B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel.

In *IEEE Int. Conf. on Automated Software Engineering (ASE'0)*. IEEE Computer Society Press, September 2000.

[Pascal *et al.*, 2008] Raymond Pascal, Roux Yvan, and Jahier Erwan. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008, 2008.

[Ployette *et al.*,] Florimond Ployette, Bertrand Jeannet, and Thierry Jéron. Stg: a symbolic test generation tool for reactive systems.

Algorithmic Complexity Attacks : the case of sorting

Quentin Ricard
Magistère student
Grenoble, France
qricard@inria.fr

Supervised by: Cédric Lauradoux and Jean-Louis Roch.

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

We investigate on *algorithmic complexity attacks* on sorting algorithms to mount denial-of-service.We focus on sorting algorithms and we show how difficult it is to generate worst case input for several sequential and distributed algorithms including `heapsort`, `introspective sort`, `hyper quicksort`. Then, we quantify the impact of algorithmic complexity attacks in terms of CPU consumption, number of comparison and permutation.

1 Introduction

A denial-of-service (DoS) attack is an attempt to disrupt services or networks connected to the Internet so they remain unavailable for their users. There are many ways to perform DoS but we focus on studying algorithmic complexity attacks. Complexity attacks consists in forcing the worst case execution of an algorithm by controlling its inputs. Doing so, an adversary can force an algorithm to consume an excessive amount of resources (CPU time or memory). Algorithmic complexity attacks were formally introduced and put into practice in [Crosby and Wallach, 2003]. In their paper, they present new attacks on hash table implementations for two versions of Perl and for the SQUID web proxy. McIlroy's paper [McIlroy, 1999] is often considered to be the first paper which have initiated algorithmic complexity attack. He presents an attack on the standard C's quicksort which is the starting point of our work. We decided to go further and investigate on distributed sorting algorithm.

`Qsort` is known to be $O(n\ log\ n)$ in average but the worst case is $O(n^2)$. McIlroy [McIlroy, 1999] explains a simple adversary on the standard C `quicksort` function based on a deterministic pivot choice. In fact recall that the three steps of `quicksort` are as follows:

- Pick an item as pivot (generally the median-of-three values in the input)
- Split the array into three parts that contain respectively all items less than the pivot, the pivot, and all items greater than the pivot.
- Recursively call `quicksort` on the sub-arrays

Standard C's `quicksort` takes an input and a comparison function as parameters, the comparison function is used to sort the input. McIlroy engineered a special comparison function that computes an input on the fly depending on how the standard `quicksort` behave. Sorted with `quicksort` the input obtained force `quicksort` to run in quadratic time.

In fact, the worst case of `quicksort` is when the pivot is always compared smaller against the items of the input. At initial state, the input contains n items evaluated to the same value. Those items as referred as "gas" in the original paper. The goal is to freeze each items into a definite "solid" value by calling `quicksort` with the special comparator. The comparator takes two input items and return weather one of them is greater or equal than the other. When the comparator is asked to compare two "gas" items, it chooses one of the items and freezes it into a value larger than all the previous frozen values and returns the comparison result of the two items (see fig. 1). At phase c), two gas items are compared, one is frozen. The other item is kept and considered as a pivot candidate: if the next comparison uses the same item as the candidate it freezes the pivot into a definitive value (phase d)). Finally, the algorithm ends and the initial input is now an input that forces `quicksort` to run on its worst case. Thus, when running `quicksort` with this input (fig. 1 from e) to h)) we can see that the maximum amount of comparison is done (exactly 5 comparisons) which correspond to the $0.25 \times n^2$ in [McIlroy, 1999]. It's worth to mention that McIlroy is a main contibutor for sorting algorithms in popular software libraries and was the first to exhibit an algorithm to build `quicksort` worst case and programming it. The purpose of this paper is to provide the same sort of tool for other sorting algorithms.

2 Sequential sorting algorithms

Many sorting algorithms have been proposed in the past (see [Cormen *et al.*, 2001] for more details). We focus in this section on two sequential algorithms such as `heapsort` and `introspective sort`. We also worked

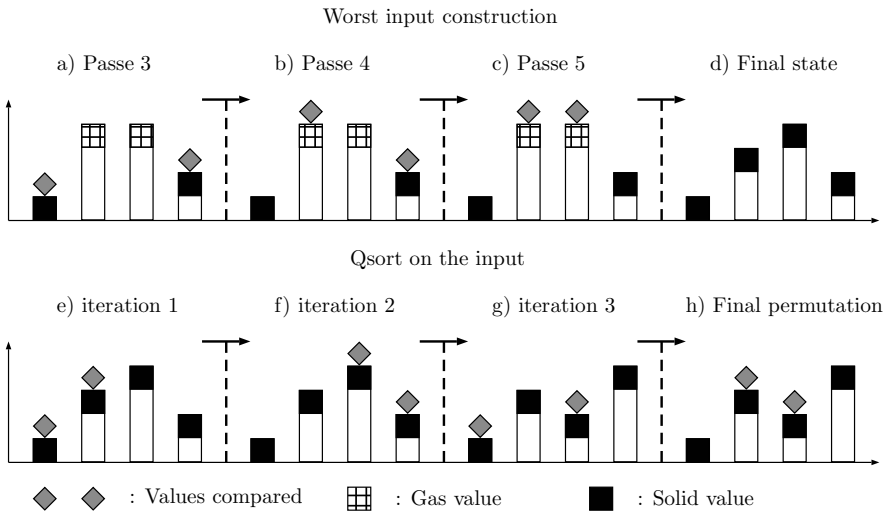


Figure 1: Killer adversary construction and execution

previously on others sequential algorithm like `bucketsort` and `mergesort` (see [Q. Ricard, 2015]). The attack on `bucketsort` helped us realize that sometimes the number of comparisons or permutations is not the good metric when considering DoS. In fact, for `bucketsort` growing the tree of recursive calls was more efficient than focusing on the number of comparison. For `mergesort`, it was really difficult to find a good adversary because the merging phase takes multiple steps and it is hard to know which combination is the most costly. The idea of this paper is to find new metrics by studying others algorithms. We do not give all the details of the algorithms. We briefly overview them, their pseudo-code can be found in appendices A and B.

2.1 The algorithms

Heapsort

We introduce `heapsort` as it is mandatory to understand `introsort`. `Heapsort` is a classic sorting algorithm first due to Williams [Williams, 1964] and improved by Floyd [Floyd, 1964]. The idea is to sort an input viewed as a complete binary tree by maintaining it heap ordered. Heap order is defined by the fact that for an input `a[1..n]` each item `a[i]` is greater `a[2i]` and `a[2i+1]`. Those positions corresponds to indexes of the children of `a[i]` when representing a tree into an input. `Heapsort` is performed in place so the input is divided in two parts one is the heap and the other represents the sorted input. Therefore, the algorithm sorts the input by removing the largest item (which is `a[1]`) of the heap by switching its position in the input with the last item. Thus, the heap is shrunk and the sorted input grows starting from its end. When moving an item outside the heap, it is possible that the new item in `a[1]` does not respect the heap order. To deal with this eventuality, the algorithm starts a repairing phase to restore the heap order (often called `siftdown` phase). Those two steps are repeated until the heap is empty and then the input is sorted.

Introspective sort

`Introspective sort` or `introsort` is an amelioration to `quicksort` due to Musser [Musser, 1997]. In fact `quicksort` worst-case complexity is $O(n^2)$ and Musser determined that it is due to the depth of the recursive calls to `quicksort`. Therefore, Musser defined a depth threshold ($2 \times log(n)$) where the algorithm switches to `heapsort` whose worst case is in $O(nlog(n))$. Thus, Musser added to `quicksort` a simple test on the current depth, when this value is zero, the subarray is sorted using `heapsort`. By adding this threshold the complexity in the worst case of `introspective sort` becomes $O(nlog(n))$ and it keeps the average-case complexity of `quicksort`. It is an interesting sort to study because of the combination of two different algorithm. This technique is often used to provide good running time. For example, in practice `quicksort` calls `insertion sort` when the number of items left in a partition is less than 15. It is due to the fact that the average case of insertion sort when considering little inputs has a better running time than `quicksort`. Thus, we though that it will be a great idea to study a combined sort and see if the worst case of such a sort can be engineered considering only the worst case of the sorts used in the combination.

2.2 Worst case datasets

Our goal is to find a list of integers such that the computation time of the algorithm is maximized. In Appendix appendix C, we provide examples of such lists for the considered sorting algorithms. Moreover, our adversaries are computed 100 times with inputs containing $1million$ integers on c programs compiled using gcc (4.4.7) on a virtual 32 cores CentOS(6.6)x86 machine.

Heapsort

`Heapsort` has the interesting property that its worst case and best case are in $O(nlog(n))$ [Schaffer and Sedgewick, 1993]. Therefore, one may asks why trying a denial of service against a non quadratic algorithm. The point is that even

if both cases are in the same complexity the actual number of comparisons can vary by a certain factor. When considering large amount of data this factor can have a huge impact on performance. Sedgewick and Schaffer [Schaffer and Sedgewick, 1993] established that the worst case depended on the distance that the `siftdown` operation has to travel to restore the heap order. They quantified the maximum amount of data moves to be no larger than $O(nlog(n)) + O(n)$. Because each moves involves two comparisons (one for picking one of the children, and one against the selected child and the integer to `siftdown`). The idea of the adversary is to inductively build the input from n integers to n+1. We use the property that to restore heap order the `siftdown` step keeps traveling down the heap until the order is restored. Thus, we can construct recursively the heap so that each time we swap two integers during the first phase the `siftdown` travels the heap until the last integer is reached. The only way to ensure this property holds, is that each node of the tree has to be inferior to one of its children. Once done with these phases, we invert the input so that the heap order is not respected. Doing so `heapsort` has to do an heapification phase before starting the sorting phase. The following schemes table 1 and table 2 explain how the scenario is built and executed by the algorithm.

Value to insert	SiftDown	SiftDown	Final State
0			0
1	**1 0**		1 0
2	**2 1** 0		2 1 0
3	**3 2** 1 0	3 **1** 2 **0**	3 0 2 1
4	**4 3** 0 2 1		4 3 0 2 1

Table 1: Design of a bad input.

Heap	SiftDown	SiftDown	SiftDown	input	Comments
1 2 0 3 4	1 2 0 3 4	**2** 1 **0** 3 4	2 **4** 0 3 **1**	Empty	heapification
4 2 0 3 1	**4** 2 0 3 **1**			4	sorting
1 2 0 3	1 2 0 3	**2** 1 0 3	2 **3** 0 **1**	4	heapification
3 2 0 1	**3** 2 0 **1**	1 2 0		3 4	sorting
1 2 0	1 2 0	2 1 0		3 4	heapification
2 1 0	**0** 1			2 3 4	sorting
0 1	0 1	1 0	**0**	1 2 3 4	Heapification and sorting

Table 2: Sorting an adversary input. The boxes around the integers represents the comparisons between the first integer and the two others. The bold integers represent the integers that have been swapped after the comparison (`heapification` phase) or the integers that are swapped (`sorting` phase).

The results – Unfortunately our adversary led the algorithm to approximately do only 1.003 times more comparisons than a randomly chosen inputs table 3 but the randomly chosen input do way more permutations. Moreover the time consumption of the random input is greater than our adversary. Therefore, we can conclude that the number of comparison is not the good metric for `heapsort`. A good idea would be to try to maximize the number of permutations. A recent work of [Suchenek, 2015] presents a complete analysis of the worst case of `heapsort`. His paper contains an algorithm to produce the worst case in terms of number of comparisons. The idea is to produce the worst case of `heapsort` by decomposing its operations. In fact he discovered that the `sift down` and the *removeMax* operations can be reversed in order to produce a bad input. He also mention strategies in order to produce bad cases for the operations that cannot be reversed. We did not manage to implement his work in order to compare with our attack.

	Adversary	Random
Average number of comparisons	3029860	3019663
Average number of permutations	1557728	1574620
Average CPU time spent (ms)	10.02	15.1

Table 3: Experimentations with an adversary input.

Introspective sort

`Introsort` is designed to switch to `heapsort` when the worst case of `quicksort` is reached. Since that `quicksort` performs better in average than `heapsort` the first thing we need to do to slow `introsort` down is to force the bad case of `quicksort` early in the partition phases so that it switches to `heapsort`. Then if we apply the worst case we found on `heapsort` we obtain an adversary for `introsort`. To do so we implemented a function like McIlroy's [McIlroy, 1999] killer for `heapsort` based on our attack on `heapsort`.

Therefore, since `introsort` switches to `heapsort` when the height of the recursive calls to `quicksort` reaches $2 \times log(n)$. Thus, we added a parameter to `killqsort` which corresponds to the depth where we our algorithm switches from building the input according to the worst case of `quicksort` to building the worst case of `heapsort`.

The results– Even if our attack on `heapsort` was not efficient it turned out to be efficient against `introsort`. The table 4 holds the results of our tests. We tested Mc Ilroy's input and also our attack without the part attacking `heapsort`. It showed good results in terms of denial of service. In fact we manages to consume almost 4 times more CPU and did 2 calls to `heapsort` where one was really expensive because it contained 99696 elements. Which means that almost all the input was sorted by `heapsort` whose average case has a greater running time than `quicksort`. The Mc Ilroy's attack only consumed twice CPU as a randomly chosen input.

3 Distributed sorting algorithm

We now discuss about a distributed sorting algorithm due to [Wagar, 1987]. There exist two improvements of this sorting algorithm. The first is the sample sort [Shi and Schaeffer, 1992] and the second hyksort [Sundar *et al.*, 2013]. They will be part of a future work and won't be discussed in this paper.

	Adversary (with heap)	Adversary (w/o heap)	McIlroy's	Random
Average number of comparisons	6601630	4289030	1864120	1915074
Average number of permutations	4949828	2352699	806802	829510
Average call to `quicksort`	164	13778	20624	20659
Average call to `heapsort`	2	113	0	0
Average CPU time spent (ms)	52.3	23.5	24.1	13.34

Table 4: Experimentations for different adversaries.

3.1 Hyper Quicksort

`Hyper quicksort` is a distributed version of the `quicksort` algorithm designed to run on D-dimensional hypercube system architecture due to [Wagar, 1987]. Initially the host processor splits the integers to all the 2^D nodes of the system. Each node sorts their N sized chunks using `quicksort`. Then the leader (node 0) pick a pivot (the median of its values) and broadcast it to the nodes in the hypercube. At this point the Hypercube is split into two equal sub-cubes, thus the nodes 0 to $2^{D-1}-1$ are neighbor to the nodes 2^{D-1} to 2^D-1. The lower half sub-cube (0 to $2^{D-1}-1$) sends its integers that are greater than the pivot to their neighbor while the upper half sub-cube (2^{D-1} to 2^D-1) sends its integers that are lesser than the pivot. This process is repeated recursively on the two sub-cubes until the sub-cubes are 0-dimensional (one single node). It has the complexity of $O(NlogN + \frac{D(D+1)}{2} + DN)$ [Wagar, 1987], where N is the initial number of integers per node, and D the dimension of the hypercube. The figure fig. 2 exposes the behavior of the algorithm.

- At iteration 1 the leader is the node A and elect 3 as pivot. As a result the hypercube splits in two and exchange happens between the nodes that are neighbors to each other. Therefore at iteration 1, A exchange with E, B with F and so on.
- Thus at iteration 2 we now have two hypercube (A, B, C, D) and (E, F, G, H) and the leaders are respectively A and E. At this stage the exchanges happen between A and C, B and D and so on.
- Finally at iteration 3, there are four leaders (A, C, E, G) with the corresponding hypercubes (A, B), (C, D), (E, F), (G, H). After the exchange phase the nodes send their items to the initial node (A).

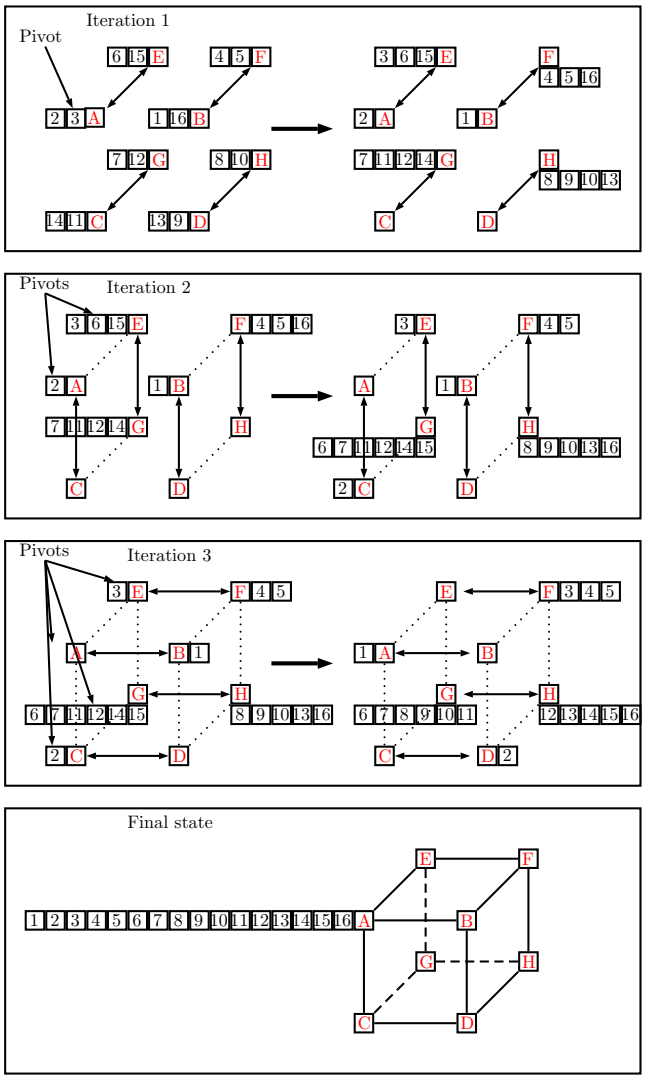


Figure 2: Hyper quicksort

3.2 Worst case dataset

When considering `hyper quicksort` one may consider the fact that the goal of distributed algorithms is to do as much instructions as possible concurrently. Thus, we design our attack with the idea to lead the algorithm to do most of the work serially. Recall that at each stage a node sends its integers to its neighbor according to whether they are greater or lesser than the pivot it received. Therefore, we designed an algorithm that, given a target T corresponding to an index of a node in a hypercube H_{yc} of dimension D. The algorithm builds an input of $(2^D \times N)$ unique integers so that, at stage D, `hyper quicksort` drives a maximum amount of integers to the node T.

Proposition 1 – The maximal number of integers in a hypercube of dimension d, where each node is initiated with N

integers, is defined as follows:

$$\begin{aligned} W(1) &= \frac{1}{2} \times N + N \\ W(d) &= \frac{W(d-1)}{2} + 2^{d-2} \times N \\ &= \frac{W(1)}{2^{d-1}} + N \times \left(\frac{2^{d+1} - 2^{3-d}}{3} \right) \end{aligned} \tag{1}$$

Proof 1 – Consider the case of a one-dimensioned hyper-cube H_{yc} where the two nodes (A and B) are initiated with N integers. Therefore the total number of integers in the hyper-cube is $2^1 \times N$.

$$\forall a \in A,\ \forall b \in B,\ a > b \tag{2}$$
$$\forall a \in A, \forall b \in B,\ a < b \tag{3}$$

Then, at stage 1 of `hyper quicksort`, which is the last stage of the sort in a one-dimensional hyper-cube, there are two possibilities :

2 $\Rightarrow$ The number of integers in A is : $\frac{N}{2} + N$

3 $\Rightarrow$ The number of integers in B is : $N + \frac{N}{2}$

Because in either case A will chose a pivot and therefore, A must send to B half of its integers. But B will send all its integers (if (1)), or none of them (if (2)). Thus we can deduce that $W(1) = \frac{N}{2} + N$.

To ease understanding our proof, we consider now the same function but for a two-dimensional hyper-cube. Thus we have four nodes $A,\ B,\ C$ and D. And the following, where $\forall I_i,\ I \in \{A,\ B,\ C,\ D\}$, and $i \in \{1,\ 2\}$ correspond to the number of integers in I at stage i. :

- At the beginning, $\forall a \in A,\ \forall b \in B,\ \forall c \in C,\ \forall d \in D$:

$$a > c > b > d \tag{4}$$
$$d > b > c > a \tag{5}$$
$$c > b > d > a \tag{6}$$
$$b > d > c > a \tag{7}$$

- At stage 1 :
  - (4) $\Rightarrow A_1 = \frac{N}{2} + N;\ B_1 = 2 \times N;\ C_1 = \frac{N}{2};\ D_1 = 0$
  - (5) $\Rightarrow A_1 = \frac{N}{2} + N;\ B_1 = 2 \times N;\ C_1 = \frac{N}{2};\ D_1 = 0$
  - (6) $\Rightarrow A_2 = \frac{N}{2};\ B_1 = 0;\ C_1 = N + \frac{N}{2};\ D_1 = 2 \times N$
  - (7) $\Rightarrow A_2 = \frac{N}{2};\ B_1 = 0;\ C_1 = N + \frac{N}{2};\ D_1 = 2 \times N$
- At stage 2 :
  - (4) $\Rightarrow A_2 = \frac{A_1}{2} + B_1;\ B_2 = \frac{A_1}{2};\ C_2 = \frac{C_1}{2};\ D_2 = \frac{C_1}{2}$
  - (5) $\Rightarrow A_2 = \frac{A_1}{2};\ B_2 = B_1 + \frac{A_1}{2};\ C_2 = \frac{C_1}{2};\ D_2 = \frac{C_1}{2}$
  - (6) $\Rightarrow A_2 = \frac{A_1}{2};\ B_2 = \frac{A_1}{2};\ C_2 = \frac{C_1}{2} + D_1;\ D_2 = \frac{C_1}{2}$
  - (7) $\Rightarrow A_2 = \frac{A_1}{2};\ B_2 = \frac{A_1}{2};\ C_2 = \frac{C_1}{2};\ D_2 = D_1 + \frac{C_1}{2}$

Thus, by simplification, whether (4), (5), (6) or (7), the nodes $A_2,\ B_2,\ C_2,\ D_2$ contain the same amount of integers which is : $\frac{W(1)}{2} + D \times N$, where $D = 2$. We just proved that $W(1) = \frac{N}{2} + N$ and $W(2) = \frac{W(1)}{2} + 2 \times N$. Let us suppose that this property holds true for $W(d-1)$. Therefore we shall demonstrate that $W(d) = \frac{W(d-1)}{2} + 2^d \times N$ is true.

$$\begin{aligned} W(d) &= \frac{w(d-1)}{2} + 2^{d-1} \times N \\ &= \frac{1}{2} \left[\frac{W(d-2)}{2} + 2^{d-2} \times N \right] + 2^{d-1} \times N \\ &= \frac{1}{2^2} \left[\frac{W(d-3)}{2} + 2^{d-3} \times N \right] + N \times \left(2^{d-3} + 2^{d-1} \right) \end{aligned} \tag{8}$$

Thus, if we continue to get back to $W(1)$ we will have the following :

$$\begin{aligned} W(d) &= \frac{W(1)}{2^{d-1}} + N \times \left(2^{d-1} + 2^{d-3} + .. + 2^{n-(2 \times n-3)} \right) \\ &= \frac{W(1)}{2^{d-1}} + 2^d N \times \left(\frac{1}{2} + \frac{1}{2^3} + ... + \frac{1}{2^{d-3}} \right) \\ &= \frac{W(1)}{2^{d-1}} + 2^d N \times \frac{2 - 2^{3-2d}}{3} \end{aligned} \tag{9}$$

Thus, 1 holds true and the maximum amount of integers that can be driven in a node at iteration is W(d) where N is the amount of integers initially contained by the nodes in the hypercube of dimension d. Now, we need to prove that it is possible to create an input of unique integers that respects the formula 1. Thus, we need to prove the following:

Proposition 2 – For any D-dimensional hyper-cube and for any node T in the hyper-cube, it exists an input Adv that at stage D of `hyper quicksort` the number of integers contained in T is maximum.

$$\begin{aligned} &\forall D \in \mathbb{N} \text{ and } \forall T \in 2^D \\ &\exists Adv \text{ such that at stage } D \\ &|T| = \frac{W(1)}{2^{D-1}} + N \times \left(\frac{2^{D+1} - 2^{3-D}}{3} \right) \end{aligned} \tag{10}$$

Proof 2 – Consider a 3-dimensional array $run[node][stage][neighbor]$ where $\forall node \in 2^D$, $\forall stage \in D$ and $\forall neighbor \in 2^D$. Given a target, run contains for each node, the list of neighbors at each stage of the execution of `hyper quicksort`. Thus, we can build the input according to the behavior of `hyper quicksort`. In fact, given a pivot p and two neighbors A(half with highest values) and B(half with lowest values) we have the following :

if $\forall i \in NA[i] < p$ then A sends all its values to B (11)

if $\forall i \in NB[i] > p$ then B sends all its values to A (12)

Therefore, if we have 11 and not 12, then, after the send/receive operations, A will be empty and $|B| = 2 \times N$.

The following example of the 3-dimensional hypercube in fig. 2 with the node A as target. Thus we establish the corresponding receive links at each stage. Applying `hyper quicksort` behavior we can create those links. We represent them on the following table :

Stage	A	B	C	D	E	F	G	H
1	E	F	G	H	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	C, G	D, H	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	B, F, D, H	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 5: Table representing the list of the node from where the integers need to come to maximize the amount present in target A.

Since we focus A the sub-cube of interest is the lower half (A, B, C, D), thus we are interested on sending only to those nodes that is why we have they are linked to E, F, G, H at stage 1 and E, F, G, H are not linked. In fact we focus here only on the sending operations that help the sub-cube of the target to get more integers. Then at stage 2 our target is again in the lower half. Thus, we are interested only on sending to A, B. Thus C, D send their initial integers and the integers they received at anterior stages. Finally we reach the last stage where B sends to A its integers. Once we have this linked list we are able to order the values that nodes must hold thanks to the position of the target and its leader at each stage. For our example this is how it works :

s1: A is in the lower half, therefore the leader of A must hold the greatest integers $\Rightarrow A = max, max--$

s2: A is in the lower half, therefore the leader of A must hold the greatest integers, and at stage 1 A is neighbor to E therefore $\Rightarrow E = max, max--$

s3: A is in the lower half, therefore the leader of A must hold the greatest integers, and at stage 2 A is neighbor to C therefore $\Rightarrow C = max, max--$ but C at stage 1 was neighbor to G therefore $\Rightarrow G = max, max--$

Thus we have the following order that must be respected for any integers in A, B, C, D, E, F, G, H we must have :

$$A > E > C > G > \{B, D, F, H\} \tag{13}$$

And we store them in an array of size 8 where each item represent the index of the chunk to allocate to a specific node at the beginning of the algorithm. For our example the allocations of the maximum gives an array like this one :

Index	0	1	2	3	4	5	6	7
index of chunk	7	$\emptyset$	$\emptyset$	5	6	$\emptyset$	4	$\emptyset$

Table 6: Table representing the array of allocations of the maximum chunks of the input. The target node is A

The integers contained in B, D, F, H just have to be inferior to G. Thus it exists 4! different inputs that lead a maximum integers in A. The $2^3 \times N$ sized inputs are constructed simply by generating an array from 1 to $2^3 \times N$ and dividing it in 2^3 chunks then allocate the highest chunk to A then to E and so on. The problem was to prove that, for any D-dimensional hypercube an input can be generated like the example above. In fact, we need to prove that it is impossible that our method leads to a contradiction like $A > E > C > F > A$.... Which means that at initial state the node A has to hold integers strictly greater than those in E, C, F but also strictly inferior. We prove now that it is impossible given the behavior of `hyper quicksort` and our algorithm.

Algorithm 1 anti-hyper quicksort

Require: A D-dimensional hypercube
Require: A 3-dimensional array $hist$ initialized as explained
Require: A target tar
Require: A array ord with 2^D elements
1: **for** $st = 1$ to D **do**
2: $lead = getleader(tar, st)$
3: **if** $inlowhalf(tar, st)$ **then**
4: $max = allocateMaxto(ord, hist[lead][st], max)$
5: **else**
6: $min = allocateMinto(ord, hist[lead][st], min)$
7: **end if**
8: **end for**

Algorithm 2 allocateMax/Minto

Require: A array $nodes$ containing the index of the nodes to allocate
Require: A array ord with 2^D elements
Require: A max/min
1: **for** $i = 1$ to $|nodes|$ **do**
2: $ord[\ nodes[i]\ ] = max/min$
3: $max++\ /\ min--$
4: **end for**
5: **return** max/min

In fact such a contradiction means that a node must appear two times in a column of the table produced by our algorithm (see table 5). In other words it means that a node can be neighbor to another node more than one time. Or this is impossible given the behavior of `hyper quicksort`. In fact the `hyper quicksort` algorithm splits an initial hypercube in two at each stage of iteration. The links of communication of two nodes are always from a node in a sub-cube and a node in the opposite sub-cube. Once split the nodes never get in contact afterwards because the algorithm always split the sub-cubes in two and never merge two sub-cubes (except at the end of the algorithm when all the integers are sent to the node 0).

Thus, our algorithm provides a $(2^D \times N)$ sized input that will lead `hyper quicksort` to send the maximal amount of integers in a single node given in parameter. We decided to go a little further so we modified the `killqsort` function of [McIlroy, 1999] so that it orders a N sized array to force the worst case of `quicksort`. The goal is to force the worst

Number of nodes	8		16		32	
Input type	Adversary	Random	Adversary	Random	Adversary	Random
Number of comparisons	34037976	33650154	8402447	81592596	202661375	192160816
Maximal size of the messages	134039	62503	213322	60039	355136	58475
CPU time spent (ms)	263.2	246.1	509.3	477.7	1697.2	1459.1

Table 7: Experimentations of the adversary input. The nodes are initiated with 100K integers. All the number correspond to average calculated over 100 run of the algorithm on the same input.

Number of nodes	8		16		32	
Input type	Adversary	Random	Adversary	Random	Adversary	Random
Number of comparisons	515103187	513966538	603345621	588774866	687958619	652206714
Maximal size of the messages	1675488	779860	1333265	374839	1109800	182594
CPU time spent (ms)	3327.5	3103.9	3405.5	3071.1	4625.8	3977.8

Table 8: Experimentations of the adversary. The initial input holds 10M integers.All the number correspond to average calculated over 100 run of the algorithm on the same input.

case of `quicksort` on each node at the first stage (because afterwards the arrays are always sorted).

The results – We tested our algorithm on three hypercubes (8, 16 and 32 nodes) and two different size of initial input. The table 7 presents the results of the attack where the nodes are initiated with $100K$ integers each. The table 8 presents the results of the attack where the nodes are initiated with $\frac{10M}{2^D}$ where D is the dimension of the hypercube (3, 4 or 5). We noticed that the difference between the maximal size of the messages that are sent between the nodes stay approximately the same regardless of the number of nodes. We also observed that the messages length was always maximal when the considered node it the node that has been targeted by the attacker. This observation derives from the fact that at the last stage all the nodes sends their integers to the initial leader and since the target will hold most of the integers the last message sent from the node will be greater than the others messages.

4 Conclusion

We described algorithmic complexity attacks on different sorting algorithms. We have shown how certain algorithms behave when confronted with adversaries inputs. Thus, certain attacks can be used for denial of service purposes while others, for example with `heapsort`, cannot or are very difficult to attempt.

In this paper, we explored on sequential and distributed algorithms, an interesting area for future research will be to study sorting algorithms used in libraries for embedded systems. In fact embedded systems (sensors) have limited resources making the choice of sorting algorithms limited and therefore they are prone to algorithmic complexity attacks. This is where DoS attacks are interesting for the attacker because the impact will affect significantly people.

References

[Cormen *et al.*, 2001] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[Crosby and Wallach, 2003] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, August 2003.

[Floyd, 1964] Robert W. Floyd. Algorithm 245: Treesort. *Commun. ACM*, 7(12):701, 1964.

[McIlroy, 1999] M. Douglas McIlroy. A killer adversary for quicksort. *Softw., Pract. Exper.*, 29(4):341–344, 1999.

[Musser, 1997] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[Q. Ricard, 2015] C. Lauradoux Q. Ricard. Algorithmic complexity attacks: the case of sorting. *TER Masters 1 IM2AG*, 2015.

[Schaffer and Sedgewick, 1993] Russel Schaffer and Robert Sedgewick. The analysis of heapsort. *J. Algorithms*, 15(1):76–100, July 1993.

[Shi and Schaeffer, 1992] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361 – 372, 1992.

[Suchenek, 2015] Marek A. Suchenek. A complete worst-case analysis of heapsort with experimental verification of its results, A manuscript (MS). *CoRR*, abs/1504.01459, 2015.

[Sundar *et al.*, 2013] Hari Sundar, Dhairya Malhotra, and George Biros. Hyksort: A new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 293–302, New York, NY, USA, 2013. ACM.

[Wagar, 1987] Bruce Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors*, 1987:292–299, 1987.

[Williams, 1964] J.W.J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

A Complexities

The complexities are expressed in terms of number of comparisons. For `hyper quicksort` since the algorithm is distributed we consider only one node. The nodes are initiated with n values and the hypercube is of dimension D.

Algorithm	Time complexity			Space complexity
	Best case	Average case	Worst case	Worst case
Heapsort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(1)$
Introsort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$
hyper quicksort	$O(\ 2^D \times n^2 + \frac{D(D+1)}{2} + Dn\ )$	$O(\ n\ logn + \frac{D(D+1)}{2} + Dn\ )$	$O(\ n\ logn + \frac{D(D+1)}{2} + Dn\ )$	$O(\ \frac{W(1)}{2^{d-1}} + n \times \left(\frac{2^{d+1}-2^{3-d}}{3}\right)\ )$

B Algorithms

Algorithm 3 `Heapsort`

Require: input $array$ with n elements
Require: Heap size $H \leftarrow n$
Require: Index i $child$ $parent$
Require: Parent save node $tparent$
Require: $i \leftarrow H/2$
while $H > 0$ **do**
  if $i > 0$ **then**
    $i \leftarrow i - 1$
    $tparent \leftarrow array[i]$
  else
    $H \leftarrow H - 1$
    $tparent \leftarrow array[H]$
    $array[H] \leftarrow array[0]$
  end if
  $parent \leftarrow i$
  $child \leftarrow i * 2 + 1$
  while $child < H$ **and** $arr[child] <= t$ **do**
    if $child + 1 < H$ **and** $array[child + 1] > array[child]$ **then**
      $child \leftarrow child + 1$
    end if
    if $array[child] > tparent$ **then**
      $array[child] \leftarrow tparent$
      $parent \leftarrow child$
      $child \leftarrow child * 2 + 1$
    end if
  end while
  $array[parent] \leftarrow tparent$
end while

Algorithm 4 `Introspective sort`

Require: input $array$ with n elements
Require: $threshold \leftarrow 2 \times log(n)$
Require: $last$,$first$
if $threshold == 0$ **then**
  $heapsort(array[first], last - first)$
else
  $part \leftarrow partition(first, last)$
  $introsort(part, last, threshold - -)$
  $introsort(first, part, threshold - -)$
end if

Algorithm 5 `Hyper quicksort`

1. Distribute the integers evenly among the nodes.
2. Each node sorts the integers it has using quicksort.
3. Node 0 broadcast its median key K to the rest of the hypercube.
4. Breakup the hypercube into two subcubes. Each node in the lower sub-cube sends its integers whose keys are $> K$ and the nodes in upper sub-cube send their integers whose keys are $\leq K$.
5. Each node merges the integers it just received with the ones he kept so that its integers are once again sorted.
6. Repeat Step 3 to 6 on each of the two sub-cubes.
7. Keep repeating step 3 to 7 until the sub-cubes consists of one single node. At that point the hypercube will be sorted.

C Worst cases

Here are the figures representing the worst case of the algorithm that we presented.

Figure 3: **Heapsort**

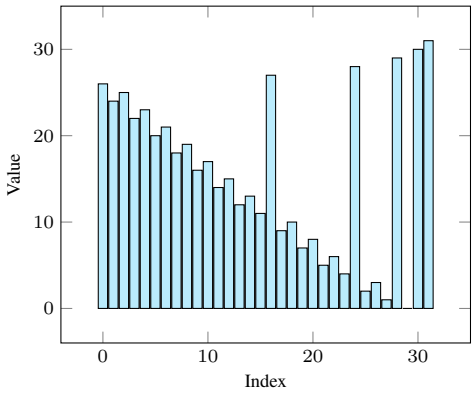


Figure 4: **Introsort**

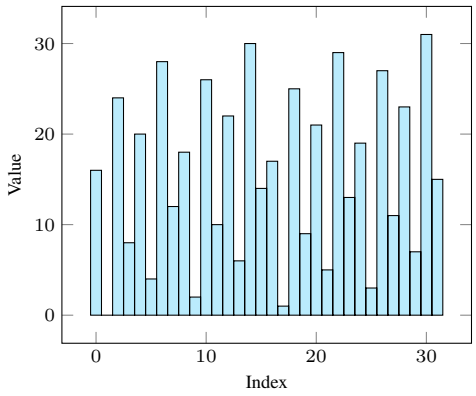
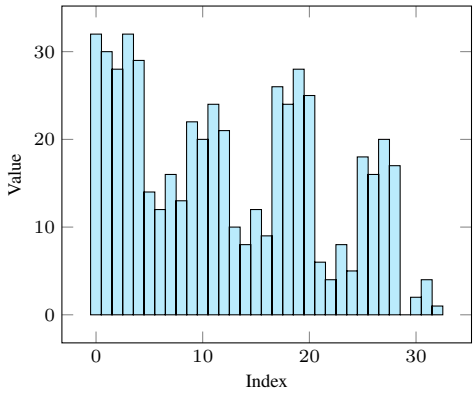


Figure 5: **Hyper quicksort (8 nodes target=7)**



Digital Humanities : computer engineers helping hellenists and latinists

My placement was divided into 2 different projects : HOMERICA and MOSCHOPOULOS

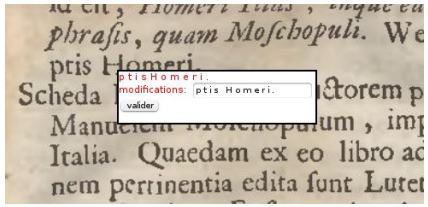
HOMERICA is a website created around 1998 and developped by Françoise Létoublon (URG 3, RARE) to gather scientific books and articles related to Homer and Ancien Greek civilization.

I was supposed to **update** it through addition of several **database**. I was especially supposed to help users to get it proper functioning.

titre	infos	desc
Conjonctures académiques ou Dissertation sur l'Ili...	François Hédelin, abbé d'Aubignac, édition critiqu...	Le pieux abbé d'Aubignac commit ce qui, aux yeux d...
Mensch - Heros - Gott Weltentwürfe und Lebensmode...	Christine Schmitz / Anja bettenworth (Hg.), 2009. ...	Das konstruktive und kritisch-reflexive Potential ...
Homers Wilder Western Die historisch-geographisch...	Heinz Warnecke, 2008. Stuttgart, Franz Steiner V...	„Das vorliegende Buch bringt einen unerhört wichti...
In the beginning was the apeiron Infinity in Gree...	Adam Drozdek, 2008. Stuttgart, Franz Steiner Ver...	The book is a historical investigation of the prob...
Les relations entre Etats dans la Grèce antique d...	Adalberto Giovannini , 2007. Stuttgart, Franz St...	Cet ouvrage n'est pas une histoire des relations i...
Libri di scuola e pratiche didattiche dall'Antich...	a cura di Lucio Del Corso e Oronzo Pecere, 2 tome...	Atti del Convegno Internazionale di Studi Cassino...
Le ragioni del sangue Storie di incesto e fratric...	Graziana Brescia, Mario Lentano, 2009. Napoli, Lo...	Negli ultimi anni la declamazione si è imposta all...
Polifonia	Diana de Paco Serrano, introducción Wilfried Floec...	
Miti e note Musicca con antichi racconti	Franco Serpa, a cura di Lorenzo De Vecchi e Corrad...	La musica è il tema intorno al quale si è costitui...

As part of digital humanities, **the Moschopoulos Projec**t is a teamwork gathering hellenists, librarians and computer engineers around a volume published in 1719. This intricated book is composed of greek commentaries of Iliad written by the Byzantine linguist Moschopoulos and of a traduction with latine notes.

Some hellenist and latinist students subjected the volume to Optical Character Recognition. Then, I had to convert it into **XML** in order to get a smart document.



I was supposed to subjected it to **Text Encoding Initiative** but it was finally removed due to technical issues. Though, I worked on **post-editing**, an additional way to correct the remaining errors of the document.

Service interétablissement de la documentation SID2 Grenoble

Maison des Sciences de l'Homme

Digital Humanities : computer engineers helping hellenist and latinist researchers

Lefrère Jules – Magistère L3 internship with Christian Boitet

My placement was divided into 2 different projects : HOMERICA and MOSCHOPOULOS

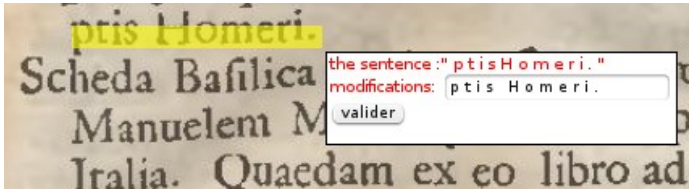
HOMERICA is a website created around 1998 and developed by Françoise Létoublon (URG 3, RARE) to gather and disseminate scientific books and articles related to Homer and the ancient Greek civilization.

I was supposed to **update** it through addition of several **databases**. I was especially supposed to help users to let it fonction properly.

titre	infos	desc
Conjonctures académiques ou Dissertation sur l'Ili...	François Hédelin, abbé d'Aubignac, édition critiqu...	Le pieux abbé d'Aubignac commit ce qui, aux yeux d...
Mensch - Heros - Gott Weltentwürfe und Lebensmode...	Christine Schmitz / Anja bettenworth (Hg.), 2009. ...	Das konstruktive und kritisch-reflexive Potential ...
Homers Wilder Western Die historisch-geographisch...	Heinz Warnecke, 2008. Stuttgart, Franz Steiner V...	„Das vorliegende Buch bringt einen unerhört wichti...
In the beginning was the apeiron Infinity in Gree...	Adam Drozdek, 2008. Stuttgart, Franz Steiner Ver...	The book is a historical investigation of the prob...
Les relations entre Etats dans la Grèce antique d...	Adalberto Giovannini , 2007. Stuttgart, Franz St...	Cet ouvrage n'est pas une histoire des relations i...
Libri di scuola e pratiche didattiche dall'Antich...	a cura di Lucio Del Corso e Oronzo Pecere, 2 tome...	Atti del Convegno Internazionale di Studi Cassino...
Le ragioni del sangue Storie di incesto e fratric...	Graziana Brescia, Mario Lentano, 2009. Napoli, Lo...	Negli ultimi anni la declamazione si è imposta all...
Polifonia	Diana de Paco Serrano, introducción Wilfried Floec...	
Miti e note Musicca con antichi racconti	Franco Serpa, a cura di Lorenzo De Vecchi e Corrad...	La musica è il tema intorno al quale si è costitui...

As part of digital humanities, **the Moschopoulos project(an AEIR project of 2013)** is a teamwork gathering hellenists, librarians and computer engineers around a volume published in 1719. This intricated book is composed of commentaries in Greek of the Iliad edited by the Byzantine linguist Moschopoulos and of a translation in Latin, also accompagned by notes in Latin.

Some hellenist and latinist students subjected the volume to Optical Character Recognition. Then, I had to convert it into **XML** in order to get a smart document.



I was supposed to transform it into an XML format conforming to the **Text Encoding Initiative(TEI)** but could not finalize that part due to technical issues. Though, I worked on **post-editing**, an additional way to correct the remaining errors in the “OCRed” document.

Linear Models sparsification for Large-Scale Text Classification

Simon Moura[*†], **Ioannis Partalas**[*], **Massih-Reza Amini**[†]

Grenoble, France

Supervised by: Ioannis Partalas, Massih-Reza Amini.

Abstract

In this work we propose a simple yet effective method for sparsifying a posteriori linear models for large-scale text classification. The objective is to maintain high performance while reducing the prediction time by producing very sparse models. This is especially important in real-case scenarios where one deploys predictive models in several machines across the network and time constraints apply on the prediction task. We empirically evaluate the proposed approach in a large collection of documents from the Large-Scale Hierarchical Text Classification Challenge. The comparison with feature selection methods and LASSO regularization shows that we achieve to obtain a sparse representation improving in the same time the classification performance.

1 Introduction

With the increasing growth of data available, the need of methods to extract automatically meaningful information becomes more and more crucial. A typical example of fast growing resources is the free online encyclopedia Wikipedia. In 2011, Wikipedia contained about $2,8$ millions articles. Since then, it received $50,000$ new articles every month and exceeded $4,5$ millions articles in early 2015.[1]

Similarly, PubMed contains more than 24 millions of medical and biomedical articles.[2] This free medical search engine receives about $500,000$ new articles every year which are assigned to one or more categories of the medical taxonomy MeSH (Medical Subject Headings). The classification of articles is still partially made by human curators. However, as an increased number of documents becomes available, the manual annotation becomes cumbersome. Towards the minimization of human effort, recent challenges focused on pushing the state-of-the-art of automatic semantic annotation of biomedical articles. BioASQ challenge is one of these initiatives[3] where participants were asked to classify new PubMed documents as they become available online, before PubMed curators classify them manually [26].

In industry, e-commerce websites such as eBay.com or Amazon face the same type of problems [9; 22]. Each day, they receive hundreds or thousands of products descriptions that should be organized in categories in order to ease the browsing in their catalog of products and allow the users to reach easily the desired information.

For browsing and accessing efficiently this large amount of collections, one needs to automatically classify these documents into a predefined ensemble of classes. In this context, the problematic of automatic classification is omnipresent. In the applications described above, one has to deal with millions of new documents every day that need to be processed in real time and to be annotated in one of the thousands of available categories. Organizing efficiently this large amount of data poses several challenges for the traditional frameworks in machine learning.

Machine learning & Classification

The task of classifying objects in a set of predefined categories has been treated thoroughly in the framework of machine learning which concerns the development of algorithms that can leverage past data for solving specific tasks.

In text classification the purpose is to automatically assign a set of articles into one of the predefined categories. As an example, consider the problem depicted in Figure 1 of classifying articles in Wikipedia .[4] Imagine that one needs to assign articles from Wikipedia in one of the following categories: *Sports*, *Technology* or *Entertainment*. To do so, one would design an algorithm that *learns* to categorize documents on a subset of Wikipedia articles using the statistical information it contains such as the number of words per article or the number of occurrence of each word. The collection of articles used for *learning*

[*]VISEO R&D

[†]Universitè Joseph Fourier

[1]en.wikipedia.org/wiki/Wikipedia:Statistics

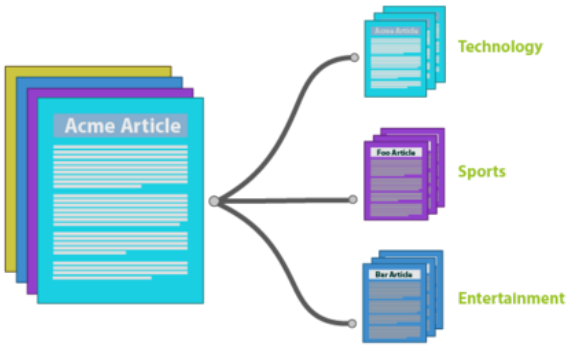
[2]http://www.ncbi.nlm.nih.gov/pubmed

[3]http://www.bioasq.org/

[4]Source: www.kdnuggets.com/2015/01/text-analysis-101-document-classification.html

is called the *training set*. In order to evaluate the performance of the algorithm designed, one would compare its classification decisions with the manual annotations on another dataset containing unseen articles called *test set*. We say that a method which classify properly unseen example generalizes well.

Figure 1: Document classification.



In machine learning, different classification tasks can be studied. We distinguish among them two main cases:

1. **Single label classification**: when it is possible to assign only one label to an object. This type of classification can itself be divided in two different cases: binary classification and multi-class classification.

   If there are only two possible labels for an object, the task concerns **binary classification**. A classical case is the classification of emails into spam or non-spam.

   On the other hand, if there are more than two possible labels for an object, it falls to the **multi-class classification** case. For instance, one could classify a news article about a *Football Player* into one of the following categories: *Sports*, *Hobby* or *People*.

2. **Multi-label classification**: when it is possible to assign more than one label to the same object. For example, a news article about *Economic crisis* can belong to both categories, *Economics* and *Politics*.

In this work we focus on single label multi-class classification.

Large-scale learning

While traditional classification tasks concerned problems with relatively few categories, in recent years new applications appeared containing tens or hundred of thousands of target classes. The inherent size of these applications poses problems for the usual machine learning algorithms.

In the context of large-scale applications, classification algorithms need to balance a trade-off among multiple factors that one seeks to optimize:

- The *training of the model*: large-scale learning involves thousands of classes and millions of characteristics. The more parameters considered, the longer will be the training phase. The main challenge here is to build algorithms that can learn on large-scale problems and produce efficient models.
- The *required memory* for storing the model. Models size have to be kept low in order to be able to process them efficiently. For instance, if models do not fit in memory, the *inference time* to classify unseen examples for large-scale problems can be largely impacted. The inference time, also called *prediction time*, is influenced my many parameters that one seek to narrow: the number of characteristics and the number of classes considered, the size of the model and the prediction algorithm used;
- The *quality of the model*, in terms of *precision*: that is how well the model performs in classifying unseen examples.

Due to the increasing amount of data available and the large number of possible categories to assign them, the problem of automatic classification of documents turns out to be difficult or even intractable in some cases. Actually, *large-scale applications* can cover multiple types of problems that need to be differentiated:

- Problems with many documents;
- Problems with a large number of characteristics;
- Problems with a large number of categories;

In this work, we focus on classification tasks which contains a large amount of characteristics and categories.

In document classification, the analysis is based on the characteristics of each document of the training set. In the general case, a characteristic, usually called *feature*, is any information that describes the object to be classified. For text classification, these features are the words contained in the collection of documents studied. A practical way to represent these features is to create a vector for each document where a coordinate corresponds to the number of occurrences of a word [21].

In a large-scale context, as the ones presented in Table 1, models can be too heavy to fit in memory. Actually, the more features and possible classes, the more computations are needed to classify an unseen example. Usually, one stores the information about each feature for each class as a real value in a huge matrix where rows represent the features and columns the categories.

Table 1 presents different statistics of the benchmark datasets from the Large Scale Hierarchical Text Challenge (LSHTC). [5]

If we consider the *DMOZ-2011* dataset which contains more than $27,000$ categories and almost 600 thousand features, one would need to store about $27,875 \times 594,158 \approx 16$ billion parameters. This translates to $123Gb$ of memory to represent the classification model.

As models of these size are hardly manageable, one needs a method to produce efficient and lightweight prediction models. There are many advantages to small models that need to be taken into account. First, they

[5] http://lshtc.iit.demokritos.gr/

Table 1: LSHTC datasets and their properties.

	#Categories	#Features	#Documents	Parameters (in GB)
DMOZ-2010	12,294	381,580	128,710	34.9
DMOZ-2011	27,875	594,158	394,756	123.4
DMOZ-2012	11,947	348,548	383,408	31
Wiki Small	36,504	346,299	538,148	94.2
Wiki Large	325,056	1,617,899	2,817,603	3918.3

are cheap to store and easier and faster to transmit for distributed prediction systems. Furthermore, the size of the model and the prediction time are closely related. In short, larger models are need more time during inference. In section 5 we compare different classification methods: while a model of $4Gb$ will take about $0,15$ seconds to classify an example, a model of $1,1Gb$ would only take $0,04$ seconds.

Purpose of this work & Contributions
This work focuses on the development of novel methods for producing sparse lightweight models for a particular set of machine learning algorithm called *linear models* that will be detailed in Chapter 2.

More specifically, we consider large-scale text classification (LSTC) tasks and target a method that should:

- Provide a similar or better performance to L_2-norm regularization for classification tasks;
- Produce very sparse models.

The size of models can be reduced using many different methods. A first idea is to reduce the size of the feature space by removing irrelevant attributes. Feature selection techniques exploit this idea by selecting a subset of relevant attributes regarding their score obtained via a given evaluation measure [14]. By reducing the number of features, we reduce the number of parameters that should be learnt and thus the size of the model.

Another possibility could be to simply remove some values in models by setting them to a null value, for instance zero. A model filled with these *null* values is called sparse. For accomplishing this, we propose a new algorithm for sparsifying a posteriori linear models. The proposed method achieves to produce very sparse models which reduces the required storage space and improves inference time. This is especially important in large-scale problems where predictive models are deployed in several machines and constraints apply on prediction time. While, such rounding techniques are not well suited for on-line methods we find that they are effective in the case of batch methods and text applications [17].

We empirically evaluate the proposed approach on a large dataset from the LSHTC [20] competition achieving not only to produce sparse models thus reducing memory requirements, but also to improve the predictive performance. We also highlight important factors for sparse models in text classification by analyzing the obtained solutions.

Part of this work has been accepted for publication and presentation in the French machine learning conference CAP (Conférence sur l'APprentissage automatique).

Organisation of the remaining of this work
Section 2 presents notation and background needed in this work. Then, in Section 3, we position our work with respect to the state of the art. In Section 4, we present the a posteriori pruning approach for large-scale linear models. In Section 5, we present experimental results obtained with our approach on a large collection of documents. In Section 6 we discuss the outcomes of this study and give some pointers to further research.

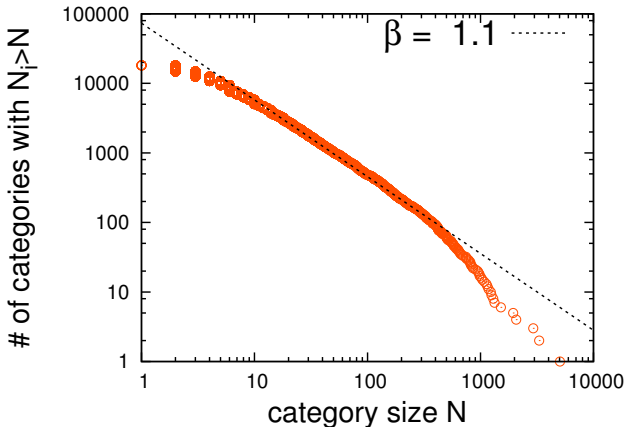
2 Challenges & Background

This section provides the theoretical background that will be used in this work. First, we define the notation and explain the challenges brought by the power-law distribution in document classification. Then, we describe the process of multi-class classification for linear models. Finally, we explain how we represent models in memory and define formally the notion of sparsity.

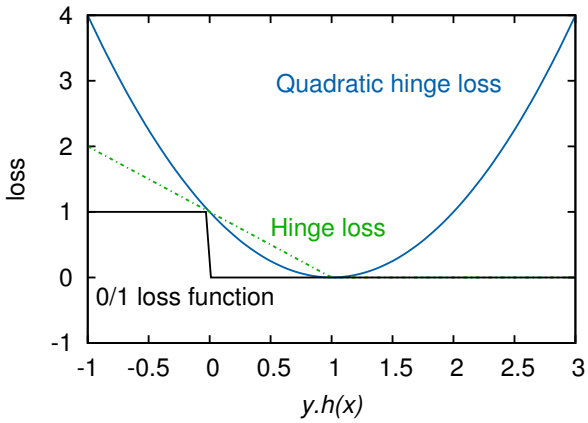
Framework
In the rest of this dissertation, $\mathscr{X}$ denotes the examples of a training set and $\mathscr{Y} = \{1,\dots,K\}$ the set of all possible outputs. $\mathbf{x} \in \mathbb{R}^n$ represents an input example obtained with the vector space model as defined by Salton [21] and $y \in \mathscr{Y}$ its associated class label, where n is the number of features and K the number of possible outputs. A document is represented by a vector in which a coordinate represents the number of occurrences of this word in the text. $\mathbf{w}_i$ denotes a classifier for class i.

2.1 Challenges of Large-Scale Multi-class Classification

Large document collections exhibit a particular property that one needs to consider in order to design efficient text classification algorithms. Specifically, it has been show that the distribution of text documents among categories is unbalanced and follow a power-law distribution [18]. In short, this law state that, in one hand, many categories have very few documents assigned to them while, in other hand, few categories have most of the documents assigned to them. This property brings particular challenges that are discussed we discuss in the following section.



(a) Distribution of 394,756 instances among 27,875 categories in LSHTC-2011. While the dataset contains less than 10 examples for more than 15,000 categories, a few categories correspond to 1000 instance or more. X axis represents the number N of documents assigned to a category. Y axis represents the number of categories with more than N documents.



(b) Example of loss functions used for classification.

Power-Law behavior of documents datasets

Figure 2a presents the category size against the rank distribution for the DMOZ-2011 dataset. More than 15,000 categories contain less than 10 examples each while a few categories contain over 1,000 examples.[6]

A power law is described as a functional relationship between two quantities, where one quantity varies as a power of the other. Formally, if N_i denotes the number of document of the i-th ranked class (in terms of number of documents), then:

$$N_i = N_1 \times i^{-\beta} \quad (1)$$

Where N_1 represents the category which contains the most documents (1st ranked category) and $\beta > 0$ denotes the exponent of the power-law distribution. This formula can be interpreted as follows: while few categories aggregate a large amount of the documents of the training set, many contain only few documents [2].

The main problem is that it is difficult to learn a good model to represent the minority classes. Indeed, only few statistical information is contained in the dataset about these categories. Thus the decision function of this categories tend to be less accurate. As a result, documents which belong to minority categories tend to be assigned to majority classes at prediction time [3].

Challenges

This particular behavior of large-scale collections lead to two main challenges:

1. For the model **training**: as explained above, the training of classification models is directly impacted by the power-law distribution of text datasets. In short, the under-representation of minority classes tend to bias the model towards the majority classes which affects directly the prediction performance.
2. For **prediction** of unseen documents: linear models are represented by a class of functions $\mathscr{H}$ of the form $\mathscr{H} = \{\mathbf{x} \mapsto \langle \mathbf{w}, y \rangle, \mathbf{w} = (\mathbf{w}_1 \dots \mathbf{w}_K)\}$. The complexity for prediction for this class of functions is $O(nK)$, where n is the number of features and K is the number of categories contained in the dataset. Considering the datasets presented in Table1, the prediction time for LSTHC-2011 dataset is $O(10^9)$ and for the large Wikipedia dataset $O(10^{12})$.

[6]The figure depicts the complementary cumulative size distribution for category sizes. It can be evaluated empirically by plotting the rank of a category's size against its size.

2.2 Background

In this section we expose the formal background related to linear classification. First, we formally define the binary classification problem in which the purpose is to classify examples when there are only two possible outputs. Then, we explain how we can extend this approach to handle multi-class classification tasks. Finally, we formally define sparsity and detail how linear models are represented in memory.

2.3 Linear Models

In several applications constraints in both space and prediction time may apply making cumbersome the maintenance of large models. For instance, for the DMOZ dataset of the LSHTC-2011 challenge that contains over 27,000 classes and over half a million of features a linear model would require approximately $124Gb$ of memory. Besides this, many features in such datasets are correlated or uninformative and can harm the performance of a predictive model.

In large-scale scenarios like for example, text classification or ad-click prediction, much attention has been

given to the deployment of linear models, mostly due to their simplicity and efficiency. In such scenarios, the vector representation of data is often sparse and the size of the feature space exceeds the size of the available training examples.

Binary classification using linear models

The purpose of binary classification algorithms is to find a function $h : \mathscr{X} \to \{0,1\}$ which map examples of the training set $\mathscr{X}$ to one of the two possible output classes $y \in \mathscr{Y}$, namely the positive class for examples labeled "1" and the negative class for examples labeled "0". The function h is called the *prediction function* or, similarly, a *classifier* and is used to *predict* the membership of an example to one class or the other.

To do so, a common approach in machine learning is to use algorithms which *learn* the function h by solving a convex optimization problem. Linear models approaches assume that the output of the classification algorithm can be expressed as a linear relation with respect to the input features.

Given a vector of inputs $\mathbf{x} = (x_1, \ldots, x_n)$, one can predict the output y_i for an example using the prediction function h:

$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 \tag{2}$$

Where the term w_0 is the intercept, also known as the *bias* in machine learning and $\mathbf{w}$ are the model parameters that one wants to learn.

Loss function

In order to fit the parameters $\mathbf{w}$ to a training set $\mathscr{X}$, a common approach is to minimize a *loss function* $\ell : \mathscr{Y} \times \mathscr{Y} \to \mathbb{R}^+$. This function measures the quality of a prediction by attributing a *cost* that compares the true label to the predicted one.

In classification three loss functions are widely used: the *0-1 loss* which compute the number of well classified examples, the *hinge loss* and the *logistic loss*.

- *0-1 loss*:

$$\ell_{0\text{-}1}(x_i) = \mathbb{1}_{h(x_i)=y_i} \tag{3}$$

- *Hinge loss*:

$$\ell_{\text{hinge}}(x_i) = \max(0, 1 - y_i h(x_i)) \tag{4}$$

  And its *squared* version:

$$\ell_{\text{hinge}}(x_i) = \max(0, 1 - y_i h(x_i))^2 \tag{5}$$

- *Logistic loss*:

$$\ell_{\text{logistic}}(x_i) = \log(1 + e^{-y_i h(x_i)}) \tag{6}$$

  Where $h(x_i)$ represents the prediction made by the decision function h for the example i and y_i is the *true* label of example i. $\mathbb{1}_{h(x_i)=y_i}$ is the indicator function which takes a value of 1 when $h(x_i) = y_i$ and 0 otherwise.

The loss functions exposed above and in Figure 2b attribute a cost of 0 to a well classified example and a positive value to misclassified example. In this context, one seeks to minimize the value of the cost function ℓ_* over the example of the dataset. In other words, one searches the a $\mathbf{w}$ which minimizes the error of prediction made on the examples of the training set.

To do so, one usually solves the convex optimization problem depicted in Equation 7 using a gradient descent algorithm:

$$J(\mathbf{w}) = \min_{\mathbf{w}} \sum_{i=1}^{m} \ell(h(x_i), y_i), \tag{7}$$

where m is the number of example contained in the training set.

A classical example of binary linear classification model is *logistic regression*. The idea of logistic regression is to decide the membership of an example by estimating the function describe in Equation 2, where the values $\mathbf{w}$ are obtained via a gradient descent algorithm. In this framework, if $h > 0.5$, we assign the example $\mathbf{x}$ to positive class 1. Otherwise we assign it to the negative class 0.

Regularization

In some cases, the hypothesis h learnt may fit well the training set, $J(\mathbf{w}) \approx 0$, but fails to generalize to unseen examples. This behavior is called *overfitting*, as the model fits too much the data. Learning algorithms, as the linear model described in Equation 2, tend to overfit the data, and thus provide poor predictive performances.

Two main reasons causes overfitting:

1. An over complex model;
2. Too many parameters considered in comparison to the number of examples in the training set.

In order to avoid overfitting and reduce the model complexity, one may use regularization methods. The idea is to constrain the value of the parameters $\mathbf{w}$ in Equation 7 in order to find a trade off between the performance on the training set and, on the other hand, the complexity of the learned function.

The problem in Equation 7 can be regularized by adding an extra term to the optimization problem. In this context, instead of minimizing the prediction error of the loss function $\ell_*(h(x_i), y_i)$, we minimize the error of the model plus the norm $\|.\|_p$ of the model itself. Formally, we want to minimize the following convex optimization problem:

$$J(\mathbf{w}) = \min_{\mathbf{w}} \sum_{i=1}^{m} \ell(h(x_i), y_i) + \lambda \|\mathbf{w}^T \mathbf{w}\|_p \tag{8}$$

where λ controls the trade off between the regularization and the fitting of the model. By setting λ to a large value, one forces the magnitude of the parameter $\mathbf{w}$ to be low. The most common variants of regularization in machine learning are L_1 and L_2 norms, but any norm can be used.

From binary to multi-class classification
Multi-class classification denotes any problem of classification that considers an output space with more than two classes, i.e. $y \in \mathscr{Y} = \{1, \ldots, K\}$ with $K > 2$.

It exists many ways to solve multi-class classification problem. Uncombined multi-class approaches like the one proposed by Crammer and Singer [11] solve directly the multi-class problem and are efficient for small datasets. However, these approaches may fail when the number of classes or the number of feature is too high.

Two other approaches have been proposed to solve the multi-class classification problem by reducing the multi-class problem to a binary one.

A first solution would be to use a *one-versus-one* approach. The idea is to construct a classifier for each couple of classes to discriminate them. For instance, if we consider a multi-class problem with three possible classes (*Sport*, *Computer Science* and *Entertainment*), one would have to learn three classifiers to discriminate the following couples:

- Sport and Technology;
- Sport and Entertainment;
- Technology and Entertainment.

In this method, $(K \times K - 1)/2$ binary problems are created, one for each pair of labels. Then a classifier is learnt for each of these pairs. The prediction is then made by a majority vote among all the classifiers. However this method is really costly due to the large number of classifiers than need to be handled during training and prediction time.

Another classical approach is One-Versus-Rest (OVR). The idea of this method is to learn K binary classifiers, one for each class of the training set [7].

Finally, to classify a new unseen example $\mathbf{x}$, one need to compute the scalar product between the new example considered with each of the classifiers of the models (one per class). Then, to choose the corresponding class for the example, one just take the maximum of these values. Formally, to classify a new example x we compute:

$$\underset{i \in [\![1,K]\!]}{\arg\max} \langle \mathbf{w}_i, \mathbf{x} \rangle + \mathbf{w}_0 \tag{9}$$

Where $\mathbf{w}_0$ represent the bias of the model and K is the number of classes considered.

The rationale of this method is that in extreme cases, OVR can be parallelized as the binary problems are supposed to be independent, while the uncombined approaches could not. This procedure is detailed in Algorithm 1.

Support Vector Machines (SVM) framework
In this work, we used a linear machine algorithm called support vector machines to learn our classifiers. SVM have been developed by Cortes and Vapnik [10] and are a state of the art linear classification algorithm for which we can use an OVR approach to solve multi-class problem.

Algorithm 1 One-Versus-Rest method for multi-class classification

Require: A linear binary classification algorithm
A training set $(x_i, y_i)_{i \in [\![1,m]\!]} \in \mathscr{X} \times \mathscr{Y}$
Where, $\mathscr{X}$ is a set of examples
And $\mathscr{Y}$ is a set of labels
for all $y_i \in \mathscr{Y}$ **do**
1. Set all examples of class y_i to label 0 in $\mathscr{X}$;
2. Set all examples of other classes to label 1;
3. Learn a binary $\mathbf{w}$ classifier to differentiate 0 (y_i) from class $y_j = 1$, $j \in \{1, \ldots K\}, j \neq i$
end for
Then to make a decision, one need to use the Equation 9 as defined previously.

In the SVM framework, one have to solve the following unconstrained optimization problem to learn the parameters vectors of the linear model:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \underbrace{\|\mathbf{w}\|_p}_{Regularization} + \underbrace{C \sum_{i=1}^{m} \ell(\mathbf{w}; \mathbf{w}_i, y_i)}_{\text{Loss function}} \tag{10}$$

Where ℓ denotes the instantaneous loss and C its regularization parameter that trades off the model complexity and the fitness of the decision hyperplane. Typical cases include the hinge loss for L_1-SVM, $\ell(y) = \max(0, 1 - t \cdot y)$, or its squared version used for L_2-SVMs, $\ell(y) = \max(0, 1 - t \cdot y)^2$, where $t = \pm 1$ regarding the real target class.

2.4 Model representation & Sparsity

Model representation
In order to store models, we usually represent its parameters as a matrix where each column represents a classifier for a given class and each row represents a feature. Thus, a value at coordinates (i, j) represents the feature i for the class j.

For instance, if we have three classes and five features, a model would look like the following matrix:

$$Model = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{pmatrix} 1.65 & 0.49 & 0.13 \\ 0.89 & 0 & 1.89 \\ 0.56 & 0.55 & 0.36 \\ 3.87 & 1.38 & 0.99 \\ 0 & 0.87 & 0.59 \end{pmatrix} \end{array} \tag{11}$$

Where f_i represents the feature i and c_j represents the class j.

Sparsity
The matrix of Equation 11 is called a *dense matrix* since the values it contains are mostly non zeros. As opposed to dense representation, *model sparsity* refers to the number of zeros which are present in the underlying linear model.

If we consider the last example11, the model contains two zeros over five features and three classes. Thus, the sparsity of this model would be $100 \times \frac{2}{15} \approx 13\%$.

More generally, if a training set have 10 classes and 100 features, the representation of the model will be a 100×10. Therefore, if this matrix have 750 zeros, we say that its sparsity is 75%.

Formally, we calculate model sparsity as follow:

$$s = 1 - \frac{\text{\# of non-zero weights}}{\text{\# of weights in model}} \tag{12}$$

3 State of the art

3.1 Sparse approaches

Sparse approaches look for models which contain a few non zero weight values. The rational is that these models are faster for inference and that they reduce the model complexity.

Aseervatham et al. [1] propose a sparse version of ridge logistic regression for altering the solution provided by logistic regression. The authors define a strictly convex optimization problem for finding a sparse solution around the ridge solution using L_1 regularization. This method is in-line with our work as it performs a posteriori the sparsification on the learned model.

Beside this, a popular method for reducing memory requirements is the use of L_1-norm as penalization term (also known as LASSO), which induces sparsity to the model [25]. Bolasso [4] is a bootstrapped version of LASSO. It selects the intersection of the set of variables selected by several replications of LASSO using bootstrap samples. In the same vein, Kim et al. proposed [15] an efficient gradient descent method for LASSO L_1 regularized that naturally shrink many features.

Langford et al. [17] introduces a method, called truncated gradient, which modifies the gradient rule in the standard stochastic gradient descent algorithm. The idea is to shrink the weights that are smaller than a predefined threshold gradually. Our method works in a similar manner but we focus on batch learning methods while inducing sparsity in the model a posteriori as we focus mainly in improving prediction time.

Koshiba et al. [16] demonstrates after extensive experiments on multiple datasets that L_1-loss SVM gives very sparse models with accuracy results close to the one obtained with a dense model obtained via the use of L_2-loss SVM.

However in applications, like in text classification, the size of the feature space exceeds the number of available examples and the features are correlated; in these cases the performance in terms of accuracy of LASSO is dominated by the use of L_2-norm [25; 30; 1]. But, L_2-norm produces dense solutions which cannot scale well in large classification problems.

The elastic nets regularization approach proposed by Zou and Hastie [30] look for a trade off between the sparsity induced by the usage of L_1-norm and the accuracy obtained by the L_2 regularization. The authors show that the regularization using both norms outperforms LASSO or ridge regression separately on various examples. Our work follow the same idea, we seek a method which enjoy both: the L_2 regularization for the accuracy and a sparse method. Nevertheless, elastic nets approach modifies *a priori* the learning of the model by the use of regularization, while are method works a posteriori on the underlying model.

Thresholding techniques have also been studied in the context of wavelets decomposition in signal processing. For a thorough treatment of multi-disciplinary sparse methods the interested reader is referred to [19].

3.2 Feature selection

Feature selection approaches aim at reducing the dimension of the input space $\mathscr{X}$ by extracting a subset of relevant features from the original dataset. The intuition is that all features are not relevant to describe a set of labels. Thus, by selecting the ones which contain useful information, we remove noise, reduce the model complexity and speed up the learning process as well as the predictive performance.

To reduce the input space, one may rely on feature selection algorithms as described by Guyon et al.[14]. An effective method for feature selection is Recursive Feature Elimination which uses an estimator (for example a (SVM)) in order to assign weights to the features. Progressively, the method selects and evaluates subsets of features. It is evident that such wrapper methods are costly for large-scale cases as the performance of the model has to be evaluated repeatedly.

In the context of feature selection, Bi et al. [6] propose a bootstrap method for selecting variables by constructing a series of linear SVMs and eliminating, after performing a linear combination of the classifiers, the variables for which the weight values do not exceed a certain threshold.

A method for feature selection for SVMs is introduced in [24]. More specifically, by introducing a binary vector which controls the selection or not of the features, the authors pose a mixed integer programming problem which is further relaxed in order to be efficiently solved.

Classical approaches for feature selection in text categorization are described by Yang and Pedersen [29]. All these methods rely on different techniques that evaluate the relevance of a feature without taking into consideration the algorithm which is used for classification. Every feature of the training set are ranked regarding a score given by one evaluation method. For instance, *Document frequency thresholding* (DF) feature selection method gives a score to each feature regarding its number of occurrences in the corpus. Then, one can decide to keep n features which were ranked the highest.

3.3 Sampling and cost sensitive approaches

In large-scale document classification, it is common to deal with unbalanced datasets where some classes are under represented. In this context, models tend to be biased towards the majority classes [5].

To overcome this problem, one solution is to re-sample the training set in order to obtain a balanced dataset and thus, to favor the minority classes. Over-sampling and under-sampling approaches work toward that idea. Over-sampling mainly consist in duplicating examples of minority classes to balance the dataset while, on the opposite, under-sampling consist in removing examples of majority classes. For further details about sampling methods, one may refer to Chawla [8].

Similarly, in order to balance the dataset, cost sensitive approaches assume that the cost of misclassification is not fixed. The idea is to use an asymmetric cost function to artificially balance the training process. In that sense, some misclassification errors may cost more than others. For instance misclassifying a document about *Basketball* in *Handball* should has a *lower cost* than misclassifying *Fried Chicken Recipe* in the *Handball* category.

Chen et al. [9] proposed to modify the loss function to balance the misclassification error over all classes and showed that the classification error can be reduced for minority classes.

In [28] Weiss et al. make an extensive comparison of cost-sensitive and sampling approaches in the framework of unbalanced datasets. They conclude that cost-sensitive, over-sampling and sub-sampling obtain close performance results in terms of precision on dataset with less than $10,000$ examples. However, if considering datasets with more examples, the cost-sensitive approach outperforms both sampling methods.

3.4 Reducing memory footprint

A different line of work concerns methods that reduce the memory footprint during the training phase of a classifier so that they can fit in memory. For instance, Golovin et al. [13] propose a method for projecting the real valued weight vector to a coarse discrete set using randomized rounding for on-line learning methods. The regret analysis show that the accumulated error during learning is small.

Recent work focused on feature hashing for reducing the memory footprint which projects the original feature space to a low dimensional space [27]. To avoid collisions and thus deteriorate predictive performance the dimension should not be decreased a lot. Finally, Takamatsu [23] propose a rounding technique for compressing the data for learning methods. While these methods compresses efficiently real-value data it does not perform any sort of sparsification.

4 Proposed method

As described in Chapter 2, in the SVM framework, one have to solve the following unconstrained optimization problem to learn the parameters vectors of the linear model:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \ \|\mathbf{w}\|_p^2 + C\sum_{i=1}^{m} \ell(\mathbf{w}; \mathbf{w}_i, y_i) \qquad (13)$$

The ridge solution (i.e. when we consider a squared loss) of this optimization problem is dense and thus cannot be used in large-scale problems were one should handle hundreds or thousands of features and classes. So, the idea is to seek a sparse solution $\hat{\mathbf{w}}$ to the optimization problem of Equation 13 that is close to the original solution $\mathbf{w}^*$. Thus, we want $\hat{\mathbf{w}}$ such that:

$$\hat{\mathbf{w}} \approx \mathbf{w}^* \qquad (14)$$

While the existing methods work a priori, the proposed approach work a posteriori, that is after training a linear model and obtaining the weights $\mathbf{w}^*$ of classifiers.

Algorithm 2 Thresholding a linear classifier.

Require: A linear classifier with weight vector $\mathbf{w}$, a threshold τ
  for $j = 1 \dots len(w)$ **do**
    if $|w_i| < \tau$ **then**
      $w_i \leftarrow 0$
    end if
  end for

In order to obtain sparse models a posteriori, one way would be to force to zero weights that are inferior to a certain threshold τ as in Algorithm 2. But this approach is too aggressive and may deteriorate the performance for larger values of τ. Thus, in Algorithm 3, we propose a trade-off between the sparsity that we aim to achieve and the value of the threshold. Larger values will lead to very sparse models but may hurt the performance by removing crucial weights. In order to ease this technique we propose to apply softer thresholding functions which sparsifies a posteriori a linear classifier by shrinking linearly the weights inferior to a second threshold ρ.

4.1 A posteriori pruning:

The general idea of the Algorithm 2 is to *remove* the values which are very low compared to the others. We simply compare the values of all weights of the model to a given threshold τ and remove them if they are lower. This algorithm has a low complexity of $O(nK)$, where n is the number of features and K the number of classes. Indeed, one only need to evaluate each value of the model once to decide whether to keep it or not.

Figure 3a depicts the effect of Algorithm 2 on the underlying model when the value of the threshold is set to $\tau = 4$. We observe that the pruning function has discontinuities in $x = \pm 4$ and that the values in $[-4, 4]$ are set to zeros, while the others remain unmodified.

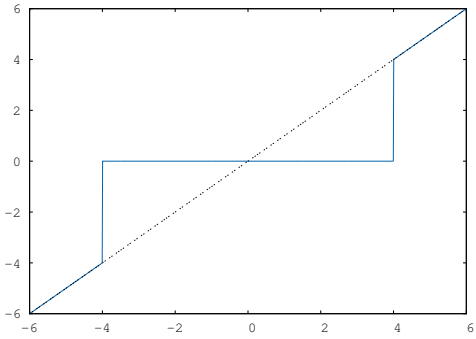
4.2 Improved pruning function

The first Algorithm 2 proposed produces a sparse representation of the underlying model, but the function used for pruning is not smooth and excessively aggressive. In Figure 3a, the function used for pruning has discontinuities and the values are either zero or the original value given by the model. The aim of the following method is: first to be less aggressive on the pruning, then to avoid discontinuities by smoothing the pruning function.

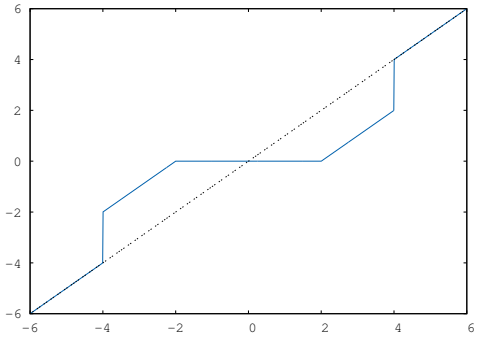
The intuition is that in text categorization problems, while small weight values may correspond to unimportant features many of them may correspond to rare features. For instance, it is the case for the minority classes which contain very few training examples. So, one should avoid discarding these weights.

In this context, the idea is that while an hard threshold as Algorithm 2 remove values if they are too small, Algorithm 3 lower down the values which are higher than a given threshold, but not high enough to remain untouched.

Algorithm 3 presents the proposed improved pruning function. The effect of this algorithm on the values of the



(a) Basic thresholding function (plain line), $\tau = 4$.



(b) Improved thresholding function, $\tau = 4$, $\rho = 2$.

Algorithm 3 Hinge thresholding of a linear classifier.

Require: A linear classifier with weight vector $\mathbf{w}$, thresholds τ and ρ
  for $j = 1 \ldots len(w)$ **do**
    if $|w_i| < \tau$ **then**
      if $w_i < 0$ **then**
        $w_i \leftarrow min(0, w_i + \rho)$
      end if
      if $w_i > 0$ **then**
        $w_i \leftarrow max(0, w_i - \rho)$
      end if
    end if
  end for

underlying model is depicted in Figure 3b.

4.3 Intuitive explanation

Intuitively these algorithm reduce the noise by removing the smallest values of the underlying model. The value of a feature in the underlying model represents its importance in the decision that will be made for classification. The highest the value of a feature for a class, the more it will influence the decision for prediction for this class.

Although a feature may be meaningful for a given class, it may be irrelevant for other classes. The idea of this algorithm is to avoid overfitting and reduce the noise created by these irrelevant features by removing meaningless values.

As said earlier, there are two main motivation behind this idea. The first one is to reduce the model size and the prediction time using a sparse representation. The second is to increase the accuracy of the underlying model by removing the noise related to the dense representation and the overfitting.

5 Experiments

This section describes the experiments we conducted in order to evaluate the proposed approach. We focused on large datasets from the text domain and more specifically we used datasets from the LSHTC challenge[7]. We present and discuss the results of the proposed approach in terms of the model sparsity, its predictive performance and its time for prediction.

5.1 Experimental setup

We evaluated the proposed method in a large-scale scenario in multi-class text classification using the 2011 DMOZ dataset of the LSHTC challenge [20]. This dataset contains 27875 categories, around $394k$ examples and $594k$ features and it is provided in a pre-processed format using stop-word removal (a list of words to filter) and stemming (taking their root form). For each document in the training set the term frequency is provided along with the assigned label. We transformed the tf vectors to the $tf * idf$ representation. We randomly sampled the DMOZ dataset with increasing number of classes.

Table 2 details the important characteristics of the sampled datasets. We note that the number of classes range from 500 to 3000 and the number of features from 68268 to 216545. It has been shown that LSTC collections generally follow a power law distribution; that is a large number of classes contain very few number of examples and most examples are contained in very small number of classes [2]. Figure 4 presents the class distribution for the 3000 classes dataset. In this case, half of the classes contain less than five examples (1309 classes).

Table 2: The main characteristics (number of classes, features and training instances) of the sample datasets used for the experiments.

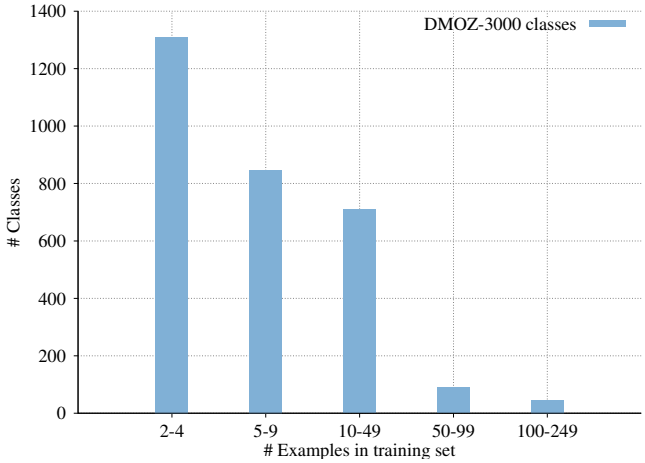
Dataset	#Classes	#Features	#Examples
DMOZ-500	500	68268	5818
DMOZ-1000	1000	104768	11123
DMOZ-2000	2000	168153	23346
DMOZ-3000	3000	216545	35533

5.2 Presentation of compared methods

As mentioned previously we work with linear classifiers and more specifically we use SVMs as our base model

[7] http://lshtc.iit.demokritos.gr

Figure 4: Class distribution in the DMOZ-3000 dataset. The minority classes are over represented.



due to its state-of-the-art performance in text classification tasks. In all experiments we used the SVM library *LIBLINEAR* to train the linear models with the squared hinge loss function [12]. We compare the following approaches on each of the samples datasets:

- L_2-SVM: L_2-regularized, L_2-loss SVM (ridge penalization).
- L_1-SVM: L_1-regularized, L_2-loss SVM (LASSO).
- χ^2: Apply the χ^2 variable selection technique at the top of L_2-SVM. The χ^2 variable selection method has been proved to be very effective for text classification [29]. It calculates a contingency table for each term t and class c and then estimates:

$$\chi^2(t,c) = \frac{N \times (AD - CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)}$$

where A is the number of times t and c co-occur, B is the number of time the t occurs without c, C is the number of times c occurs without t, D is the number of times neither c nor t occurs and N is the total number of documents.

We obtain the final score for a feature (a term in our case) by averaging its scores across the classes.

To obtain the best number of features for each datasets, we used a grid-search strategy splitting each training set in two subsets (70%/30%) and validating several percentage values for the selected features (from 5% to 70%). Table 3 reports the number of features that were selected in each dataset using grid-search.

- Our method named Linear SPARsification (LiSpar). We perform sparsification of L_2-SVM models according to Algorithm 3. In order to tune the hyper-parameters τ and ρ we relied on a simple cross-validation approach.

Table 3: Number of features kept using χ^2 and grid search. The number in parenthesis represent the percentage of features kept from the original dataset.

# Classes	# Features
500	13600 (19.92%)
1000	20953 (19.99%)
2000	42038 (24.99%)
3000	43309 (20%)

For each dataset and each algorithm we evaluated several values of the regularization parameter C ranging from 1 to 1000 . The performance of each approach is evaluated in terms of *accuracy* and *Macro F-Measure* (MaF).

Accuracy measures how often a classifier makes the correct prediction. To compute it, we used the following formula:

$$\text{Accuracy} = \frac{\text{\# of documents well classified}}{\text{\# predictions made}}$$

Considering a set of classes $\mathscr{Y} = \{1,\ldots,K\}$, the MaF is computed using the following formula:

$$MaF = \frac{2 * MaP * MaR}{MaP + MaR} \tag{15}$$

Where the macro precision (MaP) and the macro recall (MaR) are computed as:

$$MaP = \frac{\sum_{k=1}^{K} \frac{tp_k}{tp_k + fp_k}}{K} \tag{16}$$

$$MaR = \frac{\sum_{k=1}^{K} \frac{tp_k}{tp_k + fn_k}}{K} \tag{17}$$

Where tp_k, fp_k and fn_k are the true positives, false positives and false negatives respectively for class k.

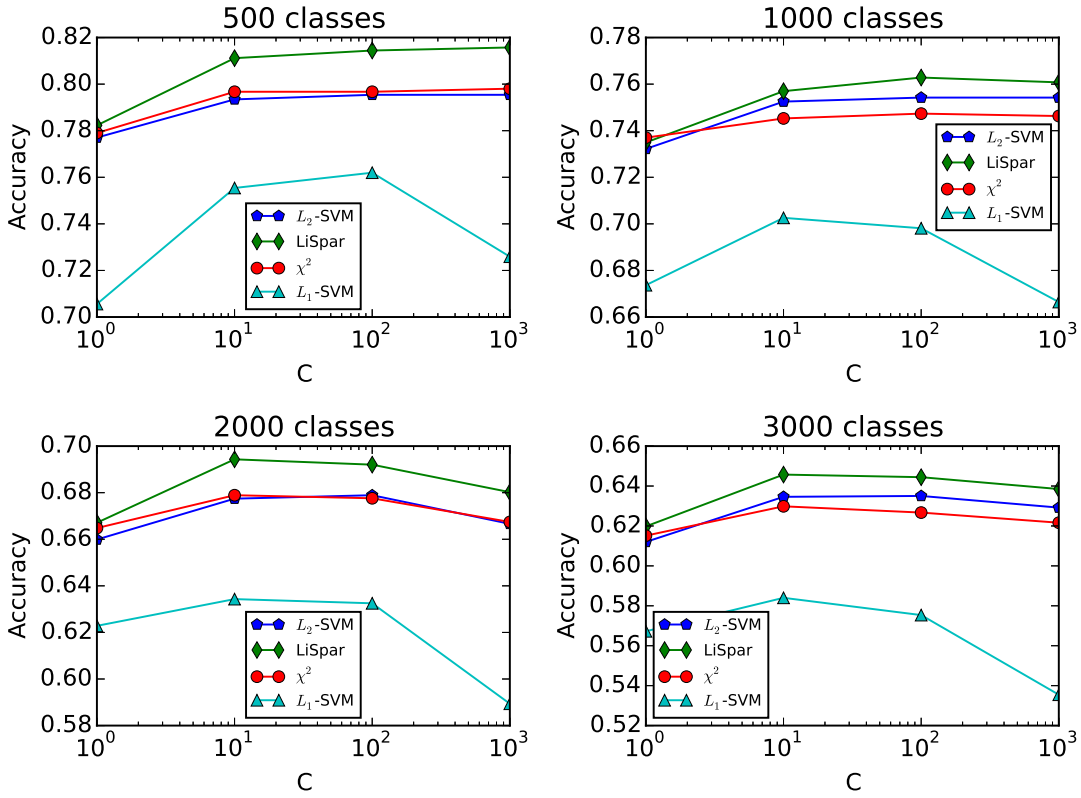
As we consider imbalanced datasets, the MaF measure is interesting since it takes into account both precision and recall and gives the same importance to minority and majority classes.

5.3 Results & Discussion

Classification Performance

Figure 5 presents the results in terms of accuracy for all competing algorithms across all datasets with respect to the penalty term C ranging from 1 to 1000 that leads to the best accuracy performance. We first observe that LiSpar outperforms all its rivals in all datasets. In most of the cases it has a stable behavior in the performance with respect to the values of the penalty term. We note that for low values of C the proposed approach will prune less values as the weight vector is bounded by the penalty term thus taking smaller values. On the other hand for larger values the weights become larger which leads to smaller percentages of pruning and in some cases to slighter improvement. LiSpar achieves its best results for parameter values C in the range of 10 to 100, where

Figure 5: Accuracy for standard SVM (L_2-regularized, L_2-loss), LiSpar (our method, SVM and threshold), χ^2 feature selection and L_1-regularized L_2-loss SVM considering regularization $C \in \{0.1, 1, 10, 100, 1000\}$ and 500, 1000, 2000, and 3000 classes



the model has not overfit the data and exhibits good performance before pruning.

L_1-SVM which produces very sparse models, hurts the performance in all cases. This is an expected behavior in large collections where the feature space exceeds the number of available examples and the features are correlated. On the other hand χ^2 maintains in most cases a similar or better performance than L_2-SVM, with the advantage of pruning a large number of uninformative features.

Figure 6 presents in the same fashion the comparison of the algorithms in terms of MaF. We can observe again a similar trend as LiSpar outperforms all its competing methods. The improvement of MaF confirms that the proposed pruning method improves particularly the classification of minority classes. Figure 7 presents the number of features that were totally pruned or not (which means that it remains at least one weight value for this feature) with respect to the term document frequency (DF) which is the number of documents a term appears in. Rare terms will have a small DF value while terms that are often used will have superior values.

Interestingly, the features that were completely pruned correspond to terms that either are rare or unimportant and thus have small values of DF. For rare terms this is a particularity for minority classes which have very few documents. On the other side for features that were not completely pruned the DF of the terms is higher and so the corresponding values in the models tend to be bigger.

Sparsity & Model size

In this section we present the sparsity ratio obtained by the proposed approach as well as the size of the model in the disk. We measure the sparsity of a model using Equation 12.

For measuring the size of the models we used a representation of similar to that of LIBLINEAR [12] for storing a dataset by keeping only the non-zero values for each class vector. Specifically, for each class we represent the vector of weights as follows:

$$\begin{aligned} &class\ 1\ f_{a_1} : v_{a_1}, \ldots, f_{b_1} : v_{b_1} \\ &\ldots \\ &class\ K\ f_{a_k} : v_{a_K}, \ldots, f_{b_K} : v_{b_k} \end{aligned}$$

where f_{a_j} and v_{a_j} are correspondingly the index and the value of feature a for the class j.

Figure 6: Macro F-Measure (MaF) for standard SVM (L_2-regularized, L_2-loss), our method (SVM and threshold), χ^2 feature selection and L_1-regularized L_2-loss SVM considering regularization $C \in \{0.1, 1, 10, 100, 1000\}$ and 500, 1000, 2000, and 3000 classes

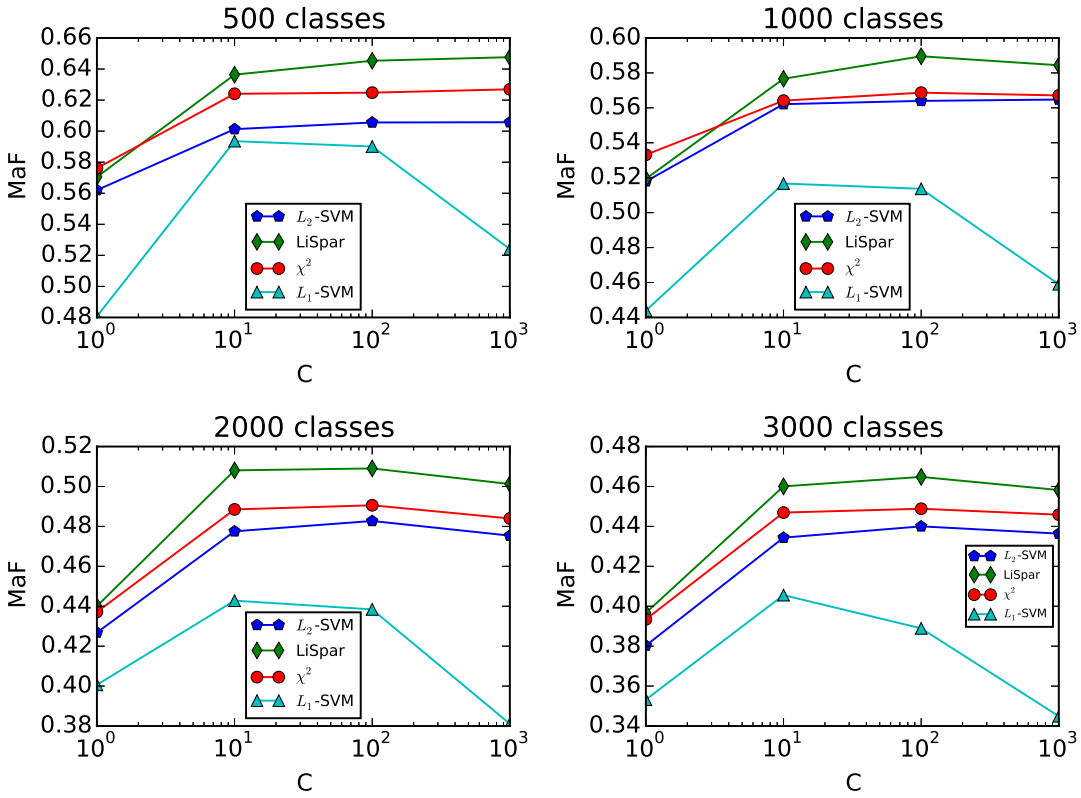


Table 4 presents the sparsity along with the model size in Megabytes for all methods across all datasets. We considered only the results for C= 100 as it usually gives the best results in accuracy. First we note that L_1-SVM achieves maximum sparsity in all cases leading to very small models. The proposed approach gives very sparse models (close to L_1-SVM) and the resulting models are at the worst case 10 times smaller than those produced by L_2-SVM. For larger number of classes, which means more features, LiSpar gives sparser models. For χ^2 as the algorithm selects informative features before the learning phase, it ends up with a lower sparsity than the other methods.

The behavior of LiSpar in terms of sparsity ratio has a two-fold implication. First, for huge datasets the models can be efficiently compressed and used in light embedded computing applications where size is critical. Second, and more importantly, it allows to reduce significantly the prediction time as one can rely on a sparse scalar product for classifying a new example.

Prediction time

In Table 5, we measured the benefit of our method in terms of prediction time compared to other approaches.

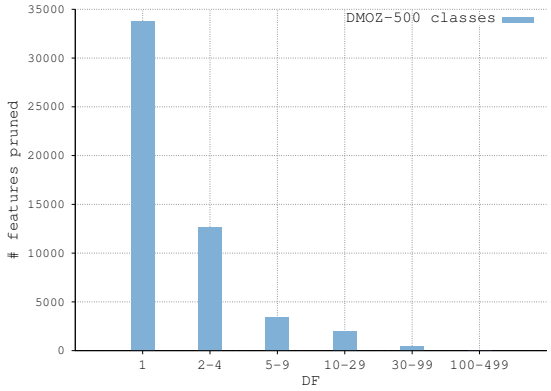
Table 5: Prediction time measurements in seconds on sparse representation for L_2-SVM, LisPar, L_1-SVM and χ^2. All models have been trained with a regularization parameter $C = 100$.

#Classes	**L_2-SVM**	**χ^2**	**LiSpar**	**L_1-SVM**
500	110.75	20.01	9.05	2.64
1000	439.01	131.94	62.9	12.71
2000	2163.43	584.17	177.74	48.39
3000	5405.64	1508.72	290.31	120.47

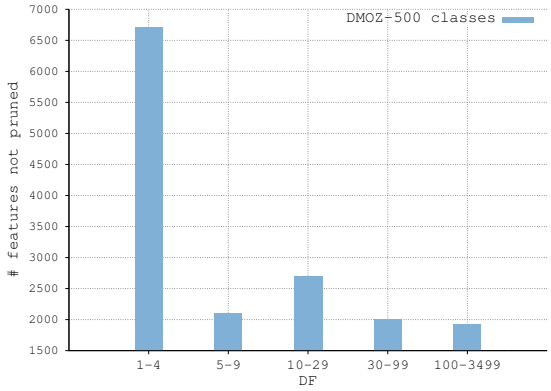
We transformed the models obtained using our different algorithms using the notation described in Section 5.3. Then we used sparse vector multiplication to compute prediction time.

The prediction time is closely related to the percentage of sparsity: the sparser models are the fastest ones. Accordingly, L_1-SVM outperform others approach in terms of prediction time. We note that the gap between LiSpar and L_1-SVM is reducing with the model size. The heavier the model, the closer they are in term of prediction time. LiSpar is 4 to 18 times faster than the standard

Figure 7: Number of features pruned with respect to their document frequency (number of documents they appear in).



(a) Fully pruned features.



(b) Not fully pruned features.

Table 4: Sparsity and models size for penalty parameter C= 100.

#Classes	L_2-SVM	LiSpar	L_1-SVM	χ^2
500	68.583% (260 Mb)	96.788% (27 Mb)	99.612% (3.1 Mb)	51.708% (72 Mb)
1000	72.479% (723 Mb)	97.062% (77 Mb)	99.701% (7.3 Mb)	57.004% (202 Mb)
2000	75.308% (2200 Mb)	98.228% (148 Mb)	99.769% (18 Mb)	66.072% (674 Mb)
3000	76.415% (4000 Mb)	99.194% (125 Mb)	99.786% (33 Mb)	66.524% (1100 Mb)

model L_2-SVM. This difference increases with the number of classes and features considered. It is also faster than χ^2 feature selection of a factor 3 to 10, which only consider a subset of the original features. Whereas χ^2 and L_2-SVM give both dense solution, χ^2 is faster than L_2-SVM as it does not consider all the features. The difference is significant as we selecting less than half of the features for each dataset.

Discussion

In short, these experiments have demonstrated that it is possible to use an a posteriori method to obtain a very sparse model while improving the predictive performance. This boost in classification accuracy is related to minority classes as we can see on MaF evaluation in Figure 6, where we got improvements up to 4% when comparing the standard L_2-SVM and our method. This improvements are even higher comparing both sparse methods: LASSO and LiSpar.

Intuitively, by removing small values from the model, we remove noise and thus small perturbations in predictions. The proposed method provides better and faster classifiers by removing the impact of irrelevant features on the decision.

Finally, such as LASSO approach, our method method provide an embedded feature selection method by reducing the number of feature considered to a small subset of the original feature space.

6 Conclusion

6.1 Conclusion

Limitations of the proposed method
Although our method improves significantly the results of underlying models in terms of accuracy, MAF and prediction time, it may still be improved.

First, LiSpar is computationally as costly as classical approach considering all features plus an overhead cost of $O(nK)$ due to the application of the pruning algorithm. Even if this overhead is insignificant regarding to the complexity of underlying algorithm, the impact may be non negligible in very large scale tasks.

Secondly, there is currently no cheap way to find optimal values for the hyper-parameters. While cross validation is a useful tool to estimate these values, it is neither: accurate nor computationally efficient.

Finally, we are still working on a formal explanation to the improvements of the proposed method in classification tasks. We have leads in regards to text classification: the pruning of weights is strongly linked to the DF and TF of terms in the collection and the terms contained in support vector tend to be less pruned than others.

Conclusion
In this work we proposed a simple approach to sparsify linear models. Whereas most approaches work a priori, this method works a posteriori and gives very sparse models while it achieves to improve the performance for text classification tasks.

Compared to other sparse approaches such as LASSO, this method gives better results in terms of accuracy and

MaF while the level of sparsity remain close. Moreover, the additional cost of this approach ($O(nK)$) is quite low compared to the standard L_2-regularized SVM approach which allow to use it on large datasets.

6.2 Future work

Based on the method proposed in this work, several directions can be investigated:

- Find a way to formally obtain the best values of the threshold's parameters investigating the loss occurred during prediction. The cross validation method is reasonable as long as the dataset used is not too large.
- Use LiSpar as a feature selection model: for instance by removing all the features for which all the weight values equal zero. This approach, called *embedded feature selection* has been studied in the framework of SVM with L_1 norm and produce very sparse lightweight models.
- As stated in the above paragraph, we are still looking for a formal explanation of the phenomena that we exposed. Starting by working on the generalization error is a first idea on which we are currently working;
- Test the same approach on different kind of datasets (e.g. images, biological datasets and so on).

7 Acknowledgments

I would like to express my gratitude to my supervisor Ioannis Partalas and Massih-Reza Amini for their kindness and their useful comments through the learning process of this master thesis. Furthermore, I want to express my thanks for the many proofreading of this work. They have both been very patient and explained to me things as long as needed.

Furthermore I would like to thank all the VISEO R&D Grenoble and LIG-AMA team for their useful discussions and their cheerful welcome.

References

[1] Sujeevan Aseervatham, Anestis Antoniadis, Éric Gaussier, Michel Burlet, and Yves Denneulin. A sparse version of the ridge logistic regression for large-scale text categorization. *Pattern Recognition Letters*, 32(2):101–106, 2011.

[2] Rohit Babbar, Cornelia Metzig, Ioannis Partalas, Eric Gaussier, and Massih-Reza Amini. On power law distributions in large-scale taxonomies. *SIGKDD Explor. Newsl.*, 16(1):47–56, September 2014.

[3] Rohit Babbar, Ioannis Partalas, Eric Gaussier, and Massih-Reza Amini. Re-ranking Approach to Classification in Large-scale Power-law Distributed Category Systems. In *ACM Special Interest Group on Information Retrieval (SIGIR 2014)*, pages 1059–1062, Gold Coast, Australia, August 2014.

[4] Francis R. Bach. Bolasso: Model consistent lasso estimation through the bootstrap. In *Proceedings of the 25th International Conference on Machine Learning*, pages 33–40, 2008.

[5] Urvesh Bhowan, Mengjie Zhang, and Mark Johnston. A comparison of classification strategies in genetic programming with unbalanced data. In Jiuyong Li, editor, *AI 2010: Advances in Artificial Intelligence*, volume 6464 of *Lecture Notes in Computer Science*, pages 243–252. Springer Berlin Heidelberg, 2011.

[6] Jinbo Bi, Kristin Bennett, Mark Embrechts, Curt Breneman, and Minghu Song. Dimensionality reduction via sparse support vector machines. *Journal Machine Learning Research*, 3:1229–1243, 2003.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[8] NiteshV. Chawla. Data mining for imbalanced datasets: An overview. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 853–867. Springer US, 2005.

[9] Jianfu Chen and David Warren. Cost-sensitive learning for large-scale hierarchical classification. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 1351–1360, New York, NY, USA, 2013. ACM.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[11] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:265–292, 2002.

[12] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[13] Daniel Golovin, D. Sculley, H. Brendan McMahan, and Michael Young. Large-scale learning with less ram via randomization. In *International Conference on Machine Learning*, volume 28 of *JMLR Proceedings*, pages 325–333. JMLR.org, 2013.

[14] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[15] Yongdai Kim and Jinseog Kim. Gradient lasso for feature selection. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 60–, New York, NY, USA, 2004. ACM.

[16] Yoshiaki Koshiba and Shigeo Abe. Comparison of l1 and l2 support vector machines. In *Neural Networks, 2003. Proceedings of the International Joint*

Conference on, volume 3, pages 2054–2059. IEEE, 2003.

[17] John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. *Journal of Machine Learning Research*, 10:777–801, 2009.

[18] Tie-Yan Liu, Yiming Yang, Hao Wan, Hua-Jun Zeng, Zheng Chen, and Wei-Ying Ma. Support vector machines classification with a very large-scale taxonomy. *SIGKDD Explor. Newsl.*, 7(1):36–43, June 2005.

[19] Julien Mairal, Francis Bach, and Jean Ponce. *Sparse Modeling for Image and Vision Processing*, volume 8 of *Foundations and Trends in Computer Graphics and Vision*. now publishers, 2014.

[20] Ioannis Partalas, Aris Kosmopoulos, Nicolas Baskiotis, Thierry Artieres, George Paliouras, Eric Gaussier, Ion Androutsopoulos, Massih-Reza Amini, and Patrick Galinari. Lshtc: A benchmark for large-scale text classification. *CoRR*, abs/1503.08581, march 2015.

[21] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *ACM Communications*, 18:613–620, 1975.

[22] Dan Shen, Jean-David Ruvini, and Badrul Sarwar. Large-scale item categorization for e-commerce. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 595–604, New York, NY, USA, 2012. ACM.

[23] Shingo Takamatsu and Carlos Guestrin. Reducing data loading bottleneck with coarse feature vectors for large scale learning. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014*, pages 46–60, 2014.

[24] Mingkui Tan, Li Wang, and Ivor W. Tsang. Learning sparse SVM for feature selection on very high dimensional datasets. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 1047–1054, 2010.

[25] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[26] George Tsatsaronis, Georgios Balikas, Prodromos Malakasiotis, Ioannis Partalas, Matthias Zschunke, MichaelR Alvers, Dirk Weissenborn, Anastasia Krithara, Sergios Petridis, Dimitris Polychronopoulos, Yannis Almirantis, John Pavlopoulos, Nicolas Baskiotis, Patrick Gallinari, Thierry Artieres, Axel-CyrilleNgonga Ngomo, Norman Heino, Eric Gaussier, Liliana Barrio-Alvers, Michael Schroeder, Ion Androutsopoulos, and Georgios Paliouras. An overview of the bioasq large-scale biomedical semantic indexing and question answering competition. *BMC Bioinformatics*, 16(1), 2015.

[27] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120, 2009.

[28] Gary Weiss, Kate McCarthy, and Bibi Zabar. Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs? In Robert Stahlbock, Sven F. Crone, and Stefan Lessmann, editors, *DMIN*, pages 35–41. CSREA Press, 2007.

[29] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 412–420, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[30] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

Improving the Performance of Multi-Tier Applications on Multicore Architectures

Hugo Guiroux
LIG, ERODS

Supervised by: Renaud Lachaize, Vivien Quéma

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

In this report, we are interested in the performance of multi-tier applications deployed on a single modern multicore machine. This context brings new challenges that need to be studied. The contribution made during this internship is twofold. First, we evaluate the impact of task placement strategies (i.e., how the application is executed on the machine) on three different use cases. The results show that the performance of the best strategies depends on different factors, such as the application architecture and the workload mix. We also manually understand the performance problems in each use case. Second, provided the difficulty to understand performance problems and the fact that the best strategies are never the same, we propose the design of a system that would dynamically mitigate performance problems of multi-tier applications. We also make preliminary validations of some key aspects of the proposed design.

1 Introduction

In this report, we are interested in multi-tier applications running on multicore machines. A multi-tier application is composed of several inter-connected applications (i.e., the tiers) exchanging messages to communicate. Each tier provides a certain functionality, and the tiers work together to perform an action. Such applications are generally server applications, where each tier is involved to generate the response to a user request. As an example, let us consider the following four-tier application. The application is composed of a Web server (tier 1 – e.g., Apache) that both serves static content (e.g., images) and dynamic content (e.g., PHP scripts), forwarding dynamic request to the dynamic content engine (tier 2 – e.g., PHP). The content engine may need to retrieve persistent data from a database (tier 3 – e.g., MySQL), while storing temporary results in an in-memory cache (tier 4 – e.g., Memcached).

Historically, to handle thousands of concurrent user requests, a multi-tier application was deployed on multiple physical machines, each one featuring a small number of cores (e.g., 2 – 4 cores). However, thanks to the evolution of hardware, commodity servers are turning into massively parallel machines (nowadays, configurations with 48 or 64 cores are common on middle-range servers) and it becomes possible to run a whole multi-tier application on a single machine. Such parallel machines are well adapted to support more and more load. Yet, the design of recent multicore machines brings new challenges. Indeed, the increasing number of cores leads to more and more complex interactions between tiers. Besides, to avoid physical limitations, the main memory is also split into several nodes (i.e., a group of cores and one or several memory banks), leading to non-uniform memory access times (NUMA): a memory access from a core located near the destination node is faster than from a remote core. This additional factor, which is not present with smaller machines, need to be considered.

As a consequence of this complexity, grasping how a multi-tier application behaves on a multicore machine, as well as troubleshooting its performance issues is a tedious process. Indeed, each tier relies on kernel mechanisms to communicate with the others, and may involve thousands of *tasks* (i.e., threads and processes) to handle all the concurrent requests. This puts a lot of pressure on the kernel, especially on the scheduler, which has the goal to decide where (i.e., on which core) and when tasks are executed, trying to improve overall performance. To control the scheduler and to help it make better decisions, one can manage the placement of tasks, allowing to restrict on which core(s) each task can be scheduled. The problem of choosing appropriate task placement strategies has been extensively studied in the context of specific application domains, such as high-performance computing applica-

tions (e.g., scientific simulations) ([Mazouz *et al.*, 2011; Zhuravlev *et al.*, 2012]). However, it has not been studied for multi-tier server applications, which have very different characteristics (e.g., lots of concurrent execution flows, variable input load across time).

The contribution made through this Master and Magistère internship is twofold. First, we use task pinning to evaluate on three workloads the impact of different task placement strategies. Task pinning is a technique that allows forcing a task to execute on a given (set of) core(s). We observe that imposing an appropriate task placement strategy can have a strong impact on performance: up to a 6x improvement compared to the default behavior of the operating system scheduler. In addition, we also highlight situations in which performance can actually be improved by decreasing the amount of CPU resources allocated to a multi-tier application (for example, we observe improvement by forcing an application to run on 18 of the 48 available machine cores). This study allows us to conclude that task placement strategies that improve performance are not always the same, and depend on several factors such as the application architecture or the workload. For each of the studied workloads, we also pinpoint the cause of the performance improvement or degradation, by leveraging kernel tracing mechanisms. This troubleshooting process shows the difficulty of finding root causes (i.e., initial causes) of performance problems in multi-tier applications where thousands of processes interact and interfere with each other.

The above mentioned facts motivate the creation of a system that would dynamically find the bottleneck tier, understand its performance problem, and mitigate it. The second contribution is the preliminary design for such a system. We also perform a set of experiments to check that some of the key design choices we propose will be viable in practice.

The rest of the report is organized as follows. Section 2 introduces the three use cases of multi-tier applications that we study, as well as our experimental testbed and our methodology. In Section 3, we evaluate the impact of different task placement strategies on our three use cases. Next, in Section 4, we analyze and understand the performance problems that we have discovered, and we design task placement strategies to mitigate them. In Section 5, we propose the design of a system mitigating performance problems of multi-tier applications, and validate some of the key choices of this design. Finally, we conclude and discuss future works in Section 6.

2 Use Cases and Experimental Setup

In this section we introduce our experimental setup. First, we present the criteria that guided our selection of use cases, as well as the injection systems. Then, we present each use case in details, including the metrics to evaluate the performance of the system under test. Finally, we present the hardware and software configurations of our testbed, as well as our experimental methodology.

2.1 Overview

Use Cases

To choose representative use cases, we took into account the following criteria: (i) realism: we are interested in server applications that have strong similarities (in terms of workloads patterns, features, design and implementation) with real-life deployments, and (ii) diversity: in order to make our conclusions as general as possible, we want to study different workloads, different web applications architectures and different kinds of performance problems. The three chosen use cases are all well-established benchmarks in industry and academia: (i) Cloudstone-A, a social-network application with a Web server, a dynamic content engine and a database (ii) Cloudstone-B, a variation of the previous use case with the addition of an in-memory cache, and (iii) SPECweb 2009 Banking, an online banking system. These three use cases are described in details in the following sub-sections.

Load Injection

A key aspect of server benchmarks is the workload model. A load injector must mimic the behavior of a real user. In a web application, a user interacts with a server system through *operations*. Each operation is composed of a request for a dynamically-generated page (e.g., a home page) and several requests for static content (e.g., pictures and CSS files). A user generally issues multiple operations in a short period of time, which corresponds to the navigation and interactions through the web application. This sequence of operations is called a session.

For all the use cases that we consider, we rely on the load-injector provided by the corresponding benchmark implementation. The workloads characteristics are derived from real-world requests traces. All these injectors are closed-loop injection systems. In a *closed-loop injection* system, a predefined number of "users" are created at the beginning of the benchmark run, and will only be destroyed at the end of the run. A user will either trigger an interaction with the server (i.e., send an operation and wait for the response), or wait for some time between two operations (called the think time). It is important to note that, in a closed-loop injection system, a new operation is only issued when the previous one is finished (and the think time is elapsed). In this model, the number of users within the system is maintained constant (except during warm-up and shutdown phases). With a closed-loop injection system, in the case of a web application, each user simulates a session, moving from page to page

using a Markov transition matrix. Each user also has a probability to end the session (i.e., the user leaves the system). However, a user that ends a session is immediately replaced with a new one. We plan to extend our study by modifying the existing benchmarks in order to consider other injection models, such as the partly-open loop [Schroeder *et al.*, 2006].

Performance Metrics

The performance metric we rely on is given by the load injection system of each use case. The metric is the throughput, expressed in *operations* (as defined in subsection 2.1) per second. We only consider operations that are valid. For an operation to be valid, all the responses (dynamic and static requests) must be correct (i.e., no error) and must comply with Quality-of-Service (QoS) constraints. QoS constraints are defined by the benchmark specification and are assessed by the load injection system that measures statistics on requests. Such constraints are, for example, an upper bound on the 95th percentile of the response time.

2.2 First Use Case: Cloudstone-A

Cloudstone [Sobel *et al.*, 2008] is a multi-tier application benchmark that implements a social-events application and aims at capturing typical functionality and behavior of Web 2.0 applications in a datacenter. This benchmark includes the Olio [oli, 2011] application as the server stack, and Faban [fab, 2015] as the load injector. We use the most recent version of Cloudstone, which is part of the popular Cloudsuite 2.0 benchmark suite [Ferdman *et al.*, 2012; clo, 2015].

The multi-tier application is composed of three tiers: (i) NGinx, (ii) PHP and (iii) MySQL NGinx is a Web server in charge of serving both static (e.g., images, CSS) and dynamic content (e.g., home page, contact form). First, the client contacts the NGinx server and requests the desired action through the HTTP protocol. When a static request arrives, NGinx reads the requested file and returns the content to the client. In this case, the request is not forwarded to any tier. On the contrary, when a dynamic request comes, NGinx acts as a proxy and forwards the request to the PHP dynamic content engine. These two tiers interact via the Fast-CGI (FCGI) [Brown, Mark R, 1996] protocol. PHP, in its turn, can interact with the MySQL database to read and/or write structured data. When it is the case (not necessarily for every input request), PHP issues one or several requests to MySQL using the MySQL protocol. Such dynamic requests may involve all the tiers, where any of them can be the bottleneck and slow down the total resolution of a client request.

2.3 Second Use Case: Cloudstone-B

The second use case is also based on Cloudstone, yet with a different configuration of the multi-tier application. This change introduces a very different behavior of the system under test, leading to two totally different results and analyses. In consequence, we consider this configuration as a distinct use case.

Compared to the first use case, the second configuration introduces an additional tier (Memcached). PHP uses Memcached for in-memory caching of temporary data. This allows lightening the burden on MySQL, putting more pressure over Memcached. Such a change, as we will see later, completely shifts the performance bottleneck of the application.

2.4 Third Use Case: SPECweb 2009 Banking

SPECweb 2009 [spe, 2009] is part of the SPEC benchmark suite, created by the Standard Performance Evaluation Corporation, which aims at standardizing benchmarks for server applications. The focus of SPECweb 2009 is to evaluate the performance of the tiers in charge of serving client requests (as opposed to back-end tiers in charge of storage and caching). For this reason, SPECweb 2009 deliberately eliminates potential bottlenecks in the back-end tiers by replacing them with a database simulator, which is an optimized program sending randomized data without computation nor synchronization. We choose to use the Banking workload, which simulates an online banking interface for account management. To support the workload, we use NGinx as the Web server, and PHP as the dynamic content engine. The database simulator is on another machine to avoid interfering with the two other tiers.

2.5 Experimental Testbed and Methodology

In this sub-section we discuss our hardware setup as well as the different software stacks and configurations used.

Hardware Setup

We used three different multicore machines as our testbed. **Machine 1** is a 48-core Dell PowerEdge R815 server with 4x AMD Opteron 6344 chips (2,6 GHz, 12 cores), and 64 GB of RAM (DDR3 1600 MHz). **Machine 2 and Machine 3** are two 64-core Dell PowerEdge R815 server with 4x AMD Opteron 6272 chips (2,1 GHz, 16 cores) has 256 GB of RAM (DDR3 1600 MHz), the second has 128 GB.

We define a *core* as a hardware processing unit that can support only one execution flow at a time In such a large machine (in terms of number of cores), the main memory is split into several *nodes*, leading to non-uniform memory access times (NUMA). Cores are split in different groups, and each group is associated with a distinct memory controller. A group of cores and the corresponding memory controller is called a NUMA node (or just node). A core can access memory from a different node by communicating through links called interconnects. On such an architecture, memory access times are not uniform: a memory access from a core located near the destination node is faster than from a remote core. Each machine is

equipped with two dual-port Intel 82599 network cards and each pair of machines is connected by two 10 Gb/s Ethernet links. For all use cases, the system under test is Machine 1. The two remaining machines are used for load injection.

Software Versions and Configuration Settings
For all the machines, we use the Ubuntu Server 12.04 operating system, with Linux kernel version 3.17.6. We use the default Linux scheduler. For Cloudstone-A and Cloudstone-B, we use NGinx version 1.7.7, PHP version 5.3.9 with the APC (PHP script cache plugin) version 3.1.9, MySQL version 5.5.20 and Memcached version 1.4.20. For SPECweb 2009 we use NGinx 1.7.10 and PHP version 5.6.6 with the embedded script cache extension. Note that all the above-mentioned software versions were either shipped with the benchmark suite (Cloudstone) or the most recent versions at the time of the experiments. We carefully tune the software parameters of each setup to achieve the best performance.

3 Performance Evaluation

In this section, through an empirical study of our three use cases, we highlight the impact of task placement strategies regarding multi-tier web applications deployed on a multicore machine. More precisely, we make the following findings: (i) the task placement strategies that yield good performance are not always the same: this depends both on the application workload and on the software architecture of the application; (ii) the most common placement strategies are not always efficient and can sometimes be pathological; (iii) in some cases, the best performance can be achieved by using only a fraction of the available CPU resources of the machine .

The section is organized as follows: first, we discuss the notion of the task placement strategy and we introduce the strategies that we consider. Second, for each use case, we describe our observations regarding the influence of task placement strategies on performance. Third, we study related works about the impact of task placement strategies. Finally, we summarize our conclusions.

3.1 Task Placement Strategies

We must first define the notion of *task*: a task is an execution flow that is schedulable by the operating system scheduler. In this work, we assume that each thread of execution at the application level (i.e., within a single-threaded or a multi-threaded process) is a task. In other words, we assume that the application-level code does not multiplex several application threads on top of a single kernel-level thread. For a set tasks, a *task placement strategy* describes where each task is allowed to be executed, i.e., on which core(s). For example, a task placement strategy for three tasks A, B and C can allow A to be executed on cores 1–6, B on cores 7–12 and C on cores 13–18.

Controlling Task Placement Through Pinning
For an application programmer or a system administrator, the main technique to control the task placement strategy is through the pinning mechanism: *pinning* a task actually means forcing this task to run on a specific set of cores. This set can either correspond to a single core (e.g., the task will only run on core 3), or multiple cores (i.e., this task can run on cores 1, 5–9, 20–26). One can also pin a task on a given NUMA node (as defined in sub-section 2.5). By default, pinning is hierarchical: if a parent task A is restricted to run only on the set of cores S, then every task created by A inherits the same restrictions. However, more complex strategies can also be implemented. For example, pinning can be used in such a way that every newly created task runs on a distinct core, assigned in a round-robin fashion (e.g., the first task runs on core 1, the second on core 2, and so on, cycling if needed).

In the context of our work, we focus on the impact of task pinning on application performance for the use cases that we have selected. When a task is pinned, several factors are impacted: (i) efficiency of the memory accesses, (ii) interferences between tasks, (iii) load balancing among cores and (iv) allocation of CPU resources . In our experiments, we have found that, among the above-mentioned factors impacted by task pinning, the allocation of CPU resources seems to have the dominant effect on the performance of the applications that we study. As a consequence, the pinning strategies that we evaluate are primarily chosen according to this factor. More precisely, pinning can be used to precisely control the amount of CPU resources allocated to a given set of tasks S. For example, pinning all the tasks of S on X nodes of a machine with N nodes will guarantee that S will never receive more than a $\frac{X}{N}$ fraction of the total CPU time. Conversely, if pinning is also used to enforce that all the other tasks be excluded from the same node, this will ensure that a $\frac{X}{N}$ fraction of the total CPU time will actually be reserved for S. We plan to take into account the other factors (in particular, the memory aspects) in future work, as discussed in Section 6.

Experimental Approach
For each use case, we evaluate the impact of different task placement strategies (obtained via pinning) on performance. We systematically evaluate three "default" pinning strategies. These strategies correspond to the typical approaches that would be considered by an experienced programmer or system administrator:

***without* pinning:** the task placement of the multi-tier application fully relies on the decisions of the kernel scheduler, which may frequently migrate tasks between cores in order to balance the load.

***node* pinning** : each task of each tier (including tasks created during the run) is allowed to be executed on one

node (e.g., for Machine 1, this corresponds to 6 cores). When a new task is created by a tier, it will be allowed to execute on the next node. Nodes are chosen in a round-robin fashion over all the nodes of the machine (e.g., the first task will be on node 0, the second on node 1). Such a pinning strategy leverages the memory affinity of a task (the task is always executed on the node where its memory is). Yet, it leaves some freedom to the scheduler, which is still able to chose between multiple cores for the task (to better balance load). This strategy is more restrictive than the *without* pinning, but less restrictive than the *core* pinning strategy (described next).

***core* pinning** : each task of each tier (including tasks created during the run) is forced to be scheduled on exactly one core. Cores are assigned in a round-robin fashion over all the cores of the machine (e.g., the first task will be on core 0, the second on core 1, ... , cycling if needed). We enforce memory and cache affinity by always executing the same task on the same core. This is the most restrictive strategy in terms of load balancing.

These three strategies use all the available nodes on the machine.

In addition to the default strategies, we evaluated many other "custom" strategies (over 15 strategies have been investigated for each workload/testbed combination). To restrain the large choices of available configuration parameters (i.e., to simplify the problem), we select pinning strategies at the granularity of a node (but a node can possibly be shared between tiers). The design of a custom strategy considers the following parameters, separately for each tier: (i) number of nodes allocated to the tier; (ii) type of allocated nodes (shared with other tiers or exclusive); (iii) type of pinning for the tasks of the tier (*without*, *core*, or *node*) . For the sake of conciseness, we only present the results for the three default strategies, as well as the best "custom" pinning strategy that we found for a specific workload and testbed.

3.2 Observations

In this sub-section, we present our experimental results for each use case.

Cloudstone-A

Figure 1a shows the results of the experiment. The best observed strategy is *m1p3n1*, where MySQL is constrained to run on one node (node 1), PHP on 3 nodes (nodes 2, 3, 4) and NGinx on one node (node 5). We can see that the *without*, *node* and *core* strategies exhibit the same performance. Finally, our best pinning strategy outperforms the others by a factor of 6.

As will be explained later (in Section 4.2)), in this configuration, MySQL is actually the bottleneck tier: its low performance has a dominating impact on the whole application behavior. And we observe that reducing the amount of CPU resources allocated to this tier improves the overall performance. Thus we can say that allocating more CPU resources to the bottleneck tier does not always improve performance and can even hurt it. This experiment also demonstrates that default (i.e., *without*, *core* and *node*) strategies are not always the best ones. We can also see that, somewhat unexpectedly, reducing the amount of CPU resources allocated to the whole multi-tier application may sometimes improve performance (using only five nodes among the eight available nodes).

Cloudstone-B

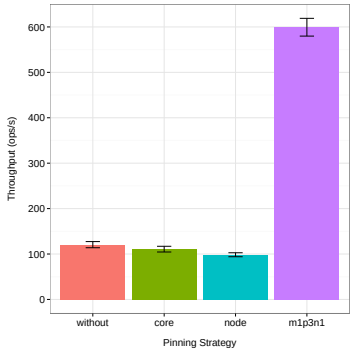
We now evaluate the performance of the Cloudstone-B use case, which is composed of four tiers: NGinx, PHP, MySQL and Memcached. Figure 1b shows performance results for Cloudstone-B. The best strategy found is *m1p8n1mem2*, where MySQL is pinned on one node (node 1), NGinx also (node 2), Memcached on two nodes (node 3, 4) and PHP is not pinned and can be scheduled on any core (relying on the decisions of the scheduler). We observe that *without* and *node* have the same performance (2580 op/s). *m1p8n1mem2* has a performance of 2845 ops/s, a 10% of performance improvement. *core*, one of our default strategies, has a performance of 2200 ops/s (14% lower than the two other defaults strategies, 22% lower than the best one). In contrast to Cloudstone-A, the default strategies does not always have the same performance.

We see that this use case works quite well with the default pinning strategy. Indeed, the best found strategy exhibits a small percentage of improvement with respect to the *without* strategy. This slight improvement can be explained by the fact that by forbidding MySQL, NGinx and Memcached to run on the same node (each one is assigned to different nodes), the scheduling interference between tiers is reduced. Indeed, when the tasks of multiple tiers are allowed to execute on the same node, the compete for hardware resources such as cores, cache space and main memory bandwidth. Moreover, as explained later (in sub-section 4.2), in this configuration, PHP suffers from CPU starvation (i.e., it does not have enough CPU resources). As a consequence, when we reduce the number of nodes allocated to the other tiers (NGinx, MySQL, Memcached), we increase the CPU resources given to PHP by the scheduler.

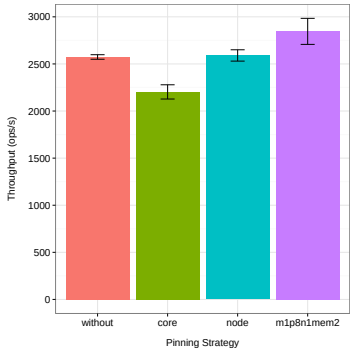
Overall, we can see that adding only one tier (Memcached) totally changes the performance as well as the behavior of the different pinning strategies. Overall, the performance impact of different task placement strategies can vary greatly according to the multi-tier application architecture, even with a similar input workload and/or similar implementations for some tiers.

SPECweb 2009 Banking

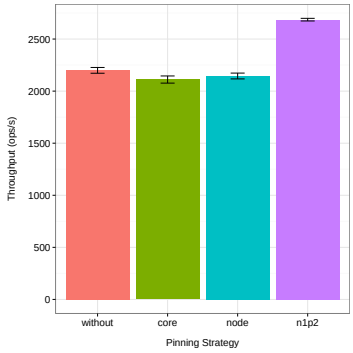
Our third use case is SPECweb 2009 Banking. SPECweb involves two tiers (NGinx and PHP) that run on Machine



(a) Comparison of pinning strategies for Cloudstone-A, injecting with 4000 concurrent users.



(b) Comparison of pinning strategies for Cloudstone-B, injecting with 18000 concurrent users.



(c) Comparison of pinning strategies for SPECweb 2009 Banking, injecting with 9000 concurrent users.

Figure 1 – Comparison of pinning strategies for the three use cases. Error bars are 95 percent confidence intervals.

1 and an external database simulator. In Figure 1c, we observe that the three default strategies yield roughly the same performance, about 2150 ops/s. The best strategy that we find is *n1p2* (NGinx on one node, PHP on two other nodes): it yields a performance of 2690 ops/s, an increase of 25% regarding the *without* strategy. In addition, it is important to notice that the latter strategy only uses three of the eight nodes available on the machine. This confirms the finding on Cloudstone-A: in some cases, the best performance can be achieved by using only a fraction of the available CPU resources of the machine. Finally, comparing results of SPECweb 2009 Banking and Cloudstone demonstrates that the best task placement strategy is dependent of the workload.

3.3 Related Work

To the best of our knowledge, our work is the first one to highlight and study in depth the strong impact of task placement through pinning for multi-tier applications deployed on a multicore architecture. Below, we discuss the main related works on the impact of task placement strategies.

The impact of pinning has been mainly studied in the context of high-performance computing (HPC) applications, and to a lesser extent for multimedia and data analysis applications [Mazouz *et al.*, 2011; Sartor and Eeckhout, 2012]. In these contexts, the studied applications generally use one task per physical core (or a small number of tasks per core). On the contrary, the web applications we are interested in generally use a great number of tasks per core in order to handle several concurrent requests and overlap the latency of input/output (I/O) operations (e.g., network and disk I/O).

The impact of task pinning and allocation of CPU resources (either for native applications or for applications encapsulated inside virtual machines) has also been studied in the context of co-scheduling distinct (and competing) applications on a shared server [Tang *et al.*, 2011; Wang *et al.*, 2012; Das *et al.*, 2013; Rao *et al.*, 2013; Blagodurov *et al.*, 2013]. In contrast, we focus on applications that are composed of several tiers. These tiers need to cooperate to serve a user request, and the amounts of execution resources needed by the different tiers are interdependent. Besides, unlike most of these works, we cannot make a distinction between interactive and background tasks in order to facilitate and prioritize scheduling decisions.

Some works have considered server applications, as we do. A part of these works only considers single-tier applications or only task placement for a single tier [Tam, David and Azimi, Reza and Stumm, Michael, 2007]. Other works consider multiple instances of the same multi-tier application on the same machine, but do not explore in depth the impact of different pinning strategies [Gaud *et al.*, 2011; Hashemian *et al.*, 2013]. Some works have also studied server applications on multicore architectures at the hardware level using the performance counters available on modern processors [Ferdman *et al.*, 2012]. They discovered inefficiencies due to mismatches between application needs and the microarchitectural features of modern processors. These works only focus on aspects related to hardware design and have not evaluated the impact of task placement. In addition, the above papers related to multi-tier applications have only studied machines with a relatively small number of cores (four to eight times lower than on the machines used in our experiments).

3.4 Summary

We have conducted a study of the impact of task placement strategies on the performance of multi-tier web ap-

plications running on multicore machines. To the best of our knowledge, our study is the first one to demonstrate the magnitude of this impact. Our main findings can be summarized as follows.

First, we saw that giving more CPU resources to the bottleneck tier does not always improve performance and can even hurt it. Then, we observed that commonly used pinning strategies (i.e., *core* or *node* pinning) may degrade performance. Besides, we observed that performance may be improved by reducing the total number of cores allocated to the application, which can be surprising. We also found that the performance of a given task placement strategy can vary greatly according to the multi-tier application architecture, even with a similar workload mix. Finally, we saw that the performance of a task placement strategy may vary according to different workload mixes (i.e., Cloudstone vs. SPECweb 2009 Banking).

With such a wide range of possibilities, we have not found a solution or a small set of solutions that always work well. In the following section, we analyze each use case to manually understand which tier is the limiting one, and find a pinning strategy that mitigates the identified issue.

4 Analysis

In this section, our objective is to understand why the different task placement strategies studied in the previous section lead to different performance. To this end, we try, for each use case, to identify which tier is the main performance bottleneck, and to understand from which performance problem it suffers.

The section is organized as follows. First, we conduct a study of related works regarding performance troubleshooting for multi-tier applications. Then, for each use case, we identify the main performance bottleneck and we propose a task placement strategy to mitigate the problem.

4.1 Multi-Tier Applications Performance Troubleshooting: Tools and Methodology

Several works have dealt with the problem of detecting, understanding and resolving performance problems for multi-tier applications.

Whodunit [Chanda *et al.*, 2007] is a tool designed to help programmers understand performance problems inside a multi-tier application. It tracks a request end-to-end through each tier using a new technique called transactional profiling. This technique allows characterizing the complete path taken into the code for each request type. Thus, the tool can detect conflicting requests types and also identify the types of user requests that trigger the heaviest work on some tiers (e.g., the application-level requests that stress the database). This kind of insight can provide guidance on modifications to optimize the design of a multi-tier application (e.g., adding a caching layer). However, this tool only allows understanding which code paths are problematic, or which requests are conflicting: it does not explain why. Moreover, it does not consider problems that are specific to multicore machines (e.g., memory controller contention), nor problems that are due to the multi-tier nature of the application, like bottlenecks induced by inefficient inter-tier traffic patterns. Besides, Whodunit requires to modify the application source code, whereas we target legacy applications.

BorderPatrol [Koskinen and Jannotti, 2008] also provides end-to-end request tracking inside multi-tier applications. Contrary to approaches like Whodunit, the application does not need to be modified. To this end, it uses a proxy in front of each tier to capture requests. Using a method called temporal join, BorderPatrol is able to correlate request relationships between tiers. The output is a trace showing how much time a request has spent in each tier. The goal of the tool is above all to trace requests through the application, while having very little information about each tier (i.e., tiers are considered as black boxes – no source code availability is required). BorderPatrol is only able to help a developer to understand where requests spend their time. Moreover, like Whodunit, BorderPatrol is not able to explain why a request spends some time in a tier (i.e., what is the performance problem).

Qingyang et al. [Qingyang *et al.*, 2011] tackle the problem of soft-resource allocation on multi-tier applications. The notion of *soft resources* refers to the resources that are configurable through each tier configuration file (e.g., thread pool size, maximum number of concurrent connections – a concrete example of soft resource, in the use cases that we study is the number of PHP processes available to handle requests). The authors demonstrate the sensitivity of soft-resource tuning: setting the allocation of some soft resources either too low or too high may hurt performance. We carefully tuned soft resources in order to avoid such problems. Besides, contrary to the authors, we consider a deployment on a single multicore machine. This introduces an additional dimension to take into account: CPU resource allocation between the different tiers. The same authors also propose an algorithm to tune the soft-resource configuration according to the needs of an application. We believe that this algorithm not sufficient for the cases that we consider. First, the authors assume that the bottleneck is necessarily due to limiting hardware resource: they do not consider bottlenecks related to software resources (e.g., lock contention). Second, this tuning algorithm only aims at maximizing the usage of the limiting hardware resource but does not consider the efficiency of this usage. For example, the CPU resource may be fully used but mostly wasted if many tasks spend time in a synchronization bottleneck (e.g., contention on a sleeping lock, or worse, on a spinlock).

Overall, none of the previously described tools meet our needs. Some of them do not take into account the

specific problems of a multicore machine and the others can only provide partial insight that is not sufficient to understand the performance bottlenecks.

4.2 Understanding Performance Bottlenecks

Most of the workloads that we study are kernel-intensive: they heavily rely on kernel mechanisms, e.g., thousands of tasks putting pressure on the scheduler, and making intensive usage of local communication channels. To understand the performance problems of these workloads, we rely on kernel tracing mechanisms. Using the Linux `perf` facility, we periodically capture kernel and user call-chain samples for the tasks of each tier. To visualize the capture, we use *FlameGraph* plots [fla, 2015]. A FlameGraph is a is visualization method that allows detecting the most frequent code paths taken by the tasks of each tier.

As a starting point of our investigations, for each use case, we generate the FlameGraph of the *without* strategy. This allows us to understand which tier has the higher CPU consumption and in which code paths each tier spends its time. Using these insights, we now explain the root cause of the performance problems observed for the two first use cases. Due to a lack of space, the analysis for the third use case is not presented in this report.

First Use Case: Cloudstone-A

Root Cause: User-Level Lock Contention We generated the FlameGraph for the Cloudstone-A use case using the *without* strategy. Due to a lack of space, we do not show this FlameGraph. On the FlameGraph, we see that most of the CPU time is consumed by MySQL, especially inside the `mutex_spin_lock` function. The name of the function gives a hint on the performance problem: MySQL seems to spend most of its CPU time trying to acquire lock(s). After investigating the MySQL source code [mys, 2015], it appears that the function is related to the `InnoDB buffer pool`. This buffer pool is an in-memory cache that InnoDB (i.e., a database storage engine used by MySQL) uses to avoid reads from the disk. We observe that concurrent accesses to this cache are synchronized with a coarse-grain locking scheme, which explains the heavy contention when MySQL is stressed.

Task Placement Strategy In this use case, MySQL is the performance bottleneck. Thus, an efficient task placement strategy must be directed towards it. The observed lock contention is caused by a too high number of concurrent accesses: it is important to understand that giving more CPU resources to the incriminated tier will only degrade performance. Indeed, the more there are concurrent execution flows trying to access the same lock, the more they will wait on it. Therefore, we need to reduce the amount of CPU resources allocated to MySQL. The number of nodes allocated should neither be too low (MySQL would suffer from CPU starvation) nor too high (there would be contention inside MySQL). In the Cloudstone-A use case, we found experimentally that one node (6 cores on Machine 1) is the right amount of CPU resources.

For the other tiers, when searching for the best configuration, we also tried to choose a strategy that allocates the minimum amount of nodes that each tier needs. This allows saving CPU resources. We have found that NGinx needs one node and PHP needs three nodes. This explains how we found our best configuration *m1p3n1* (MySQL on one node, PHP on three nodes and NGinx on one node) for Cloudstone-A, which uses only 63 percent of the total amount of available CPU resources.

Second Use Case: Cloudstone-B

Root Cause: CPU Starvation We also generate the FlameGraph for Cloudstone-B (not shown here). First of all, with the FlameGraph, we can see that the MySQL bottleneck is not present anymore. Indeed, most of the time is now spent inside PHP. This acts as evidence that Memcached reduces the load on MySQL. In order to determine if PHP, as MySQL previously, suffers from lock contention, we zoom into the PHP call-chains. As it turns out, there is no evidence that PHP suffers from a problem like this, and seems to use its CPU times efficiently.

It is not surprising that PHP consumes most of the CPU time. Indeed, most of the business logic of the request processing is done by PHP, and this becomes the most costly processing phase when MySQL is not on the critical path. Thus, we emit the hypothesis that PHP is limited by the amount of available CPU resources. To validate our hypothesis, we initially set the number of nodes allocated to each tier like in our best strategy (*m1p8n1mem2*: MySQL one node, NGinx one node, Memcached two nodes) and we vary the number of nodes allocated to PHP, observing the application performance. Then, we increase the amount of nodes allocated to PHP and see that the performance also increases. These results validate our hypothesis that the performance problem of Cloudstone-B is PHP suffering from CPU starvation.

Task Placement Strategy In order to mitigate the identified performance problem, our approach to choose a task placement strategy is in opposition with the one we considered for Cloudstone-A. As a matter of fact, we do not want to reduce the CPU resource of the bottleneck tier. Rather, we need to reduce the amount of CPU resource of all other tiers, to use the saved resource for the bottleneck tier. In the context of a multicore machine, finding the right amount of CPU resources for each tier is non-trivial. We do not want to reduce too much the allocation of a tier, otherwise it may suffer from CPU starvation. Yet, we still want to save the highest possible amount of resources for the bottleneck tier.

In the Cloudstone-B case, our best-found strategy is *m1p8n1mem2* (MySQL on one node, NGinx on one node, Memcached on two nodes and PHP is free to be scheduled on any core). We find that MySQL needs at least one node, NGinx also needs one node and Memcached needs two nodes. Finally, by allowing PHP to be scheduled on any node, we allow it to exploit as many CPU resources as possible.

Summary

For the first two use cases, we have provided an explanation for the main performance problems. We also explained how to mitigate such problems by using optimized task placement strategies. In the next section, we discuss the design of a system that could dynamically and automatically detect, understand and mitigate performance problems of a multi-tier application running on a multicore machine.

5 Towards Dynamic Optimizations

In Sections 3 and 4, we have seen that the task placement strategy that yields the best performance depends on multiple factors, and that the root cause of the performance problem is never the same. These facts motivate the need for a solution that could automatically and dynamically address the problems that we have highlighted. First of all, we provide a brief overview of the state of the art regarding dynamic optimizations for multi-tier applications, finding that no previous work fully meets our expectations. As a consequence, we propose design guidelines for a system mitigating the performance problems we are interested in, and perform experiments validating some of the key choices of this design.

5.1 Related Work

In this section, we briefly review the main existing solutions that aim at dynamically optimizing the performance of a multi-tier application.

A first solution is SEDA [Welsh *et al.*, 2001; Welsh and Culler, 2003]. SEDA introduces a new multi-tier application design, where each tier is modeled by a *stage*, composed of a pool of threads and a queue of pending requests. Each stage is managed by a controller, whose goal is to detect and react to performance problems within the stage. SEDA does not fully meet our needs with respect to several aspects: (i) it requires to completely rewrite the application, whereas we want to support legacy application, and (ii) the approach used by SEDA to mitigate the performance bottlenecks (i.e., increasing the thread pool size) may not always address the problems that we target, or may even be counterproductive (e.g., lead to CPU starvation).

A second solution is proposed by Urgaonkar et al. [Urgaonkar *et al.*, 2007; Urgaonkar *et al.*, 2008]. In this work, the authors propose a generic model to detect the performance problems of a multi-tier application. This solution is not adapted to our situation, for two main reasons. First, authors consider deployment on a cluster of physical machines. On a single multicore machine, the resource allocation of each tier is dependent of the resource allocation of every other tier. Second, we refrain from requiring a thorough application characterization. We believe that, the complexity arising from the deployment on a multicore machine does not allow to know in advance what is, for each tier, a good resource allocation policy.

None of the solutions presented above fully meets our needs. As a consequence, in the next sub-section, we propose the design of a system whose goal is to automatically and dynamically detect the bottleneck tier, understand the root cause of the performance problem and mitigate it.

5.2 Design Proposal

We propose design guidelines for a system that mitigates the performance problems of a multi-tier application. Our approach intends to be generic: it is not specific to any multi-tier application or multi-tier architecture and it works with legacy applications, relying on operating system tracing and task placement mechanisms. The architecture of our solution will be decomposed into three main components, which together perform the mitigation. The following sub-sections give more details about each component, along with the organization of their execution.

First Component: Identification of the Bottleneck Tier

The role of the first component will be to identify the bottleneck tier. To this end, we plan to model the multi-tier application using queuing theory where tiers are nodes and queues represents requests waiting to be handled. Indeed, we believe such a theory is adapted to our needs.

In models based on queuing theory, each tier has many characteristics (e.g., request processing time) that, we believe, will allow the tool to identify the bottleneck tier. To retrieve them, we plan to dynamically compute some of these characteristics by capturing and analyzing events related to inter-tier communications, and derive others using equations given from the theory.

Second Component: Understanding of the Performance Problem

The second component will be in charge of understanding from which performance problem the bottleneck tier suffers (e.g., lock contention, CPU starvation). Thus, the component will need to collect and analyze a set of system-level metrics (e.g., scheduling statistics, lock concurrency levels) in order to pinpoint the precise kind of problem. Each metric will serve two purposes: (i) decide if the bottleneck tier suffers from the performance problem detected by this metric, and (ii) validate the impact of the performance improvement modifications

made on the tier: the measured value will reflect the impact of the performed changes .

Third Component: Mitigation of the Performance Problem

The third component will be in charge of choosing a task placement strategy adapted to the mitigation of the performance problem identified by the second component, and apply this strategy to the application's tasks at runtime. When choosing a task placement strategy, the component will need to carefully balance resource allocation among the tiers, to avoid CPU starvation and reduce CPU interference.

Coordination of the Three Components

The three components need to be executed one after the other. Moreover, once the third component has modified the application task placement strategy, the system needs to execute the three components again. Indeed, modifying the performance of a tier may change the throughput of the bottleneck tier and therefore the throughput of the other tiers. A new tier may become the bottleneck, which need to be detected and handled by the system. The solution iterates the execution of the components until reaching system "stability": when there is no more performance problem that the system can mitigate. The dynamic optimization system will also have to be triggered at regular intervals in order to handle the potential variations of the input load.

5.3 Preliminary Design Validation

During the design of the system, we made design choices for each one of the three components. Before starting to implement the proposed design, it is important to verify if these choices are viable. In other words, we need to verify: (i) that it is possible to observe and/or perform the needed actions (with enough precision and reactivity) and (ii) that every action has a low overhead (i.e., it does not significantly slow down the running application).

In particular, we consider that it is necessary to validate the three following important choices. The first one is the possibility for the first component to dynamically capture the required events to feed the multi-tier model, in order to identify the bottleneck tier. The second one is the possibility for the second component to find and compute a metric that detects the performance problem of this bottleneck tier. Finally, the third one is the feasibility for the third component of applying a task placement strategy at run-time. For each one of these choices, we present an experiment validating its viability. Due to a lack of space, we only present the experiment validating the first component.

Dynamic Tracing of Interactions Between Tiers

As explained previously, we plan to model the multi-tier application using queuing theory. We consider tiers as black boxes: we have very few information on them. Yet, we expect that by feeding our model with dynamically computed input parameters, the system will be able to identify the bottleneck tier. In order to compute these parameters, we have observed requests characteristics (e.g., processing time, arrival time) inside the multi-tier application. To this end, we have captured events related to inter-tier communications (request and response production/consumption) using kernel tracing mechanisms.

Before establishing our model, we first want to verify if the capture of these events is feasible. In other words, we have checked that the capture does not significantly modify the behavior or hurt the performance of the application. Using the Cloudstone-A with the *m1p3n1* task placement strategy (the best strategy for this use case), we have compared the performance with and without the capture enabled.

The results of this experiment show less than one percent of performance overhead (without capture: 570 ops/s, with a confidence interval of 95% of the mean equals +- 19 ops/s; with capture: 565 ops/s, +- 14 ops/s). We consider this overhead as insignificant.

Even if we still need to confirm that using queuing theory will allow detecting bottleneck tiers, preliminary results show that it is feasible to dynamically capture some of the input parameters for this type of model.

6 Conclusion and Future Works

We studied the impact of task placement strategies on multi-tier applications deployed on a single multicore machine and proposed the preliminary design of a system that will dynamically mitigate performance problems of such applications.

We first introduced the multi-tier Web application use cases, our experimental testbed and the configuration choices we made for evaluating the impact of task placement strategies using pinning. Then, we studied the impact of task placement strategies on our three use cases. We observed up to a 6x improvement compared to the default behavior of the operating system scheduler using specific strategies. In addition, we also highlighted situations in which performance can actually be improved by decreasing the amount of CPU resources allocated to the multi-tier application (using roughly 40% of the total CPU resource available). From this study, we concluded that the best task placement strategies are not always the same, and depend on several factors such as the application architecture or the workload. We also pinpointed, for every use case, the root cause of the observed performance problems. This troubleshooting process showed the difficulty of understanding performance problems in multi-tier applications. Motivated by our findings, we proposed the preliminary design of a system that will dynamically identify the bottleneck tier, understand its performance problem, and mitigate it. Finally, we validated some of the key choices made during the design phase.

Acknowledgment

This work has been supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

References

[Blagodurov *et al.*, 2013] Sergey Blagodurov, Daniel Gmach, Martin Arlitt, Yuan Chen, Chris Hyser, and Alexandra Fedorova. Maximizing Server Utilization While Meeting Critical SLAs via Weight-based Collocation Management. In *Symposium on Integrated Network Management (IM)*, pages 277–285. IEEE, May 2013.

[Brown, Mark R, 1996] Brown, Mark R. Fastcgi specification. *Open Market Inc.*, 1996. Accessed: 2015-04-27.

[Chanda *et al.*, 2007] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-tier Applications. In *European Conference on Computer Systems (EuroSys)*, pages 17–30. ACM, 2007.

[clo, 2015] Cloudsuite 2.0 Benchmark. `http://parsa.epfl.ch/cloudsuite`, 2015. Accessed: 2015-04-27.

[Das *et al.*, 2013] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In *Symposium on High Performance Computer Architecture (HPCA)*, pages 107–118. IEEE, Feb 2013.

[fab, 2015] Faban, A Free and Open Source Performance Workload Creation and Execution Framework. `http://faban.org/`, 2015. Accessed: 2015-04-27.

[Ferdman *et al.*, 2012] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: a Study of Emerging Scale-Out Workloads on Modern Hardware. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48. ACM, 2012.

[fla, 2015] FlameGraphs: a visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately. `http://www.brendangregg.com/flamegraphs.html`, 2015. Accessed: 2015-06-05.

[Gaud *et al.*, 2011] Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Gilles Muller, and Vivien Quéma. Application-Level Optimizations on NUMA Multicore Architectures: the Apache Case Study. LIG Research Report RR-LIG-011, March 2011. https://hal.inria.fr/hal-00950933.

[Hashemian *et al.*, 2013] Raoufehsadat Hashemian, Diwakar Krishnamurthy, Martin Arlitt, and Niklas Carlsson. Improving the Scalability of a Multi-core Web Server. In *International Conference on Performance Engineering (ICPE)*, pages 161–172. ACM, 2013.

[Koskinen and Jannotti, 2008] Eric Koskinen and John Jannotti. BorderPatrol: Isolating Events for Black-box Tracing. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, pages 191–203. ACM, 2008.

[Mazouz *et al.*, 2011] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Performance Evaluation and Analysis of Thread Pinning Strategies on Multi-core Platforms: Case study of SPEC OMP applications on Intel architectures. In *High Performance Computing and Simulation (HPCS)*, pages 273–279, Istanbul, Turkey, July 2011. IEEE.

[mys, 2015] MySQL Server source code. `https://github.com/mysql/mysql-server`, 2015. Accessed: 2015-04-27.

[oli, 2011] Olio, a Web 2.0 Toolkit. `http://incubator.apache.org/projects/olio.html`, 2011. Accessed: 2015-04-27.

[Qingyang *et al.*, 2011] Wang Qingyang, S. Malkowski, Y. Kanemasa, D. Jayasinghe, Pengcheng Xiong, C. Pu, M. Kawaba, and L. Harada. The Impact of Soft Resource Allocation on n-Tier Application Scalability. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 1034–1045. IEEE, May 2011.

[Rao *et al.*, 2013] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng zhong Xu. Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pages 306–317. IEEE, Feb 2013.

[Sartor and Eeckhout, 2012] Jennfer B. Sartor and Lieven Eeckhout. Exploring Multi-threaded Java Application Performance on Multicore Hardware. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 281–296. ACM, 2012.

[Schroeder *et al.*, 2006] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Networked Systems Design & Implementation (NSDI)*, pages 18–30, San Jose, CA, USA, May 2006. USENIX Association.

[Sobel *et al.*, 2008] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *Workshop on Cloud Computing and its Applications (CCA)*, pages 1–6. IEEE, October 2008.

[spe, 2009] SPECweb2009: Benchmark for Evaluating Web Server Performance. `https://www.spec.org/web2009/`, 2009. Accessed: 2015-04-27.

[Tam, David and Azimi, Reza and Stumm, Michael, 2007] Tam, David and Azimi, Reza and Stumm, Michael. Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors. In *European Conference on Computer Systems (EuroSys)*, pages 47–58. ACM, March 2007.

[Tang *et al.*, 2011] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *International Symposium on Computer Architecture (ISCA)*, pages 283–294. ACM, 2011.

[Urgaonkar *et al.*, 2007] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic Modeling of Multi-tier Internet Applications. *ACM Transactions on the Web*, 1(1), May 2007.

[Urgaonkar *et al.*, 2008] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1:1–1:39, March 2008.

[Wang *et al.*, 2012] Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W. Davidson, and Mary Lou Soffa. Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 156–167. IEEE, 2012.

[Welsh and Culler, 2003] Matt Welsh and David Culler. Adaptive Overload Control for Busy Internet Servers. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 4–18. USENIX Association, March 2003.

[Welsh *et al.*, 2001] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles (SOSP)*, pages 230–243. ACM, October 2001.

[Zhuravlev *et al.*, 2012] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Computing Surveys*, 45(1):4:1–4:28, December 2012.

Using Static Single Assignament in Dynamic Binary Translation *

Antoine Faravelon
Grenoble Alpes
Grenoble, France
antoine.faravelon@imag.fr

Supervised by: Frédéric Pétrot.

Abstract

As System on a Chip are more and more complex, full system simulation becomes the only viable option to develop applications in a timely fashion for them. Yet, functional accuracy is not enough, specifically as systems embed more and more processor, and simulation speed is of utmost importance. State of the art cross compiled Instruction Set Simulators (ISS) uses Dynamic Binary Translation (DBT) to obtain good simulations speed. While it enables them to perform better than instruction by instruction interpretation, the translation phase is still naive. Indeed, it is akin to compilation with a frontend-middle end-backend structure. The middle end however has limited optimization. This report presents method to enhance performances of cross compiled ISS. To do so, the possibility to implement an SSA Intermediary Representation in DBT and its use to enhance middle end optimization abilities have been studied. It led to implement an SSA IR in a state of the art DBT ISS, Qemu. Performances were overall improved, with up to 20% gain on some benchmarks which is attributed to better host register usage. Time spent in middle and back end was measured to be negligible on most benchmarks and pretty much so even on a Linux boot.

1 Introduction

System on a Chip (SoC) nowadays are more and more complex. Developing applications for them and estimating their performance has thus grown to be more and more complicated. But, in the same time, market pressure is pushing developers to develop applications before first devices using these processors have even been fully designed. Android smartphones are a perfect example of that: QualComm has released around 80 different Snapdragon platform instances to its customers in 6 years, Texas Instruments advertised 25 during the same period, and many other players in the market have similar numbers. To permit software development ahead of hardware availability, full system simulation is the only solution. To that aim, we focus on fast execution of cross-compiled code during this internship. It consists of executing code designed for a target CPU on an already, readily available one. Although simulation accuracy is of course important, to be practically useful, speed is of the essence. The state of the art in instruction set simulation relies on Dynamic Binary Translation (DBT), which now manages to achieve good translation speed. Instead of an instruction per instruction interpretation, DBT translates each executed basic block of the target processor binary code into host binary. Most modern DBT simulators use a compiler like structure, that is, frontends which decode target code into an Intermediary Representation (IR). This IR is processed by a middle end, which can perform optimization and is then translated to host binary by the backend. However, code generation seems to be a weak point. Optimization is quite limited, thus generated code is potentially sub optimal. In particular one of the state of the art and probably the most widely used dynamic binary translator, Qemu[Bellard, 2005], only implements liveness analysis and constant folding. This means that there is a potential speed improvement to be gained there. A naive solution to obtain this gain would be to craft optimizations individually for each existing simulator. However, designing an optimization algorithm which is both fast and efficient is nothing simple and requires a thoughtful study. One solution to tackle both sides of the problem and avoid the need of such a study is to directly take inspiration from compiler techniques. In particular, specific intermediary representations such as Static Single Assignment (SSA) are widely used in compiler optimization. Their properties allow to design quick, and efficient optimization, which seems particularly adapted to DBT code generation problem as speed is of the utmost importance and yet generated code quality is sought for. For SSA, an important number of optimizations have already been designed. These may be ported to any kind of SSA IR with much less effort than designing or porting algorithms from scratch. This then seems to be the most adapted solution to answer both sides of the code generation problem. The aim of this work is to implement the SSA IR in Qemu. Using this intermediary form, it is then possible to quickly compute live ranges of variables thanks to implicit def use chains. Optimizations such as liveness analysis can then be performed in a quicker way. Code generation can thus be

*These match the formatting instructions of IJCAI-07. The support of IJCAI, Inc. is acknowledged.

accelerated as the naive liveness analysis algorithm can be quite complex in the general case. However, in SSA form it is vastly simplified as one use may only have one definition, the algorithm shall then stabilize faster. New optimizations can also be done. For example, when exiting SSA form, it becomes possible to rename variables that have disjoint live ranges as the same destination variables. Host register spilling can thus be reduced at code generation. Later on, the use of SSA could also lead to cache the intermediary form itself and use it to re-generate code as needed when, for example, hot paths are detected. However, as this IR was not designed with DBT in mind, two main difficulties related with its use still persist. Firstly, building (and memorizing) an SSA form may be too heavy for DBT. This has been tackled by identifying constraints on IR code which have been identified in DBT and then simplifying a reference, linear time, SSA construction algorithm presented in [Sreedhar and Gao, 1995]. The second difficulty is to get out of SSA without being able (or hardly) to add real copy and yet be efficient, allow for optimizations, and minimize the number of variables as in general algorithms such as [Sreedhar *et al.*, 1999] and [Boissinot *et al.*, 2009]. Another difficulty is that we may not in any way break compatibility with the existing framework. Our IR should be fully transparent to existing front ends and back ends.

2 Background

Before going further let us recall a few important notions used in the rest of this report.

2.1 Dynamic Binary Translation

DBT consists of translating a target machine code into an host machine code, on a target basic block per target basic block basis. The code thus translated is cached so as to avoid having to re-translate it every time it needs to be executed. Caching the code allows to amortize the cost of translation and potential optimization that were done on it. In all recent approaches, translation is actually made in two steps. First, the code is translated into an IR, instruction by instruction (stages "Fetch" and "Decode" in Fig. 1). When a branch instruction is found, the translation stops ("Yes" link in Fig. 1). The IR block is then slightly optimized by the middle end and, finally, translated into host code ("Code generation" stage in Fig. 1) and cached.

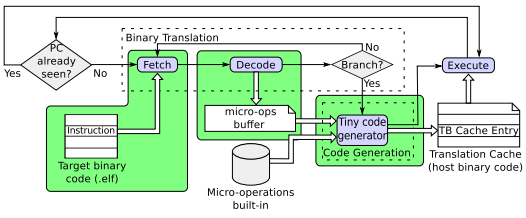


Figure 1: Dynamic Binary Translation Principle (illustration taken from [Gligor *et al.*, 2009]).

3 Static Single Assignment

3.1 Principle

Static Single Assignment is a set of properties applied to an Intermediary Representation. The fundamental property is that only one definition of any variable may exist. A definition being any kind of affectation. This property implies some interesting corollaries for optimization. Firstly, any use of a variable corresponds to one and only one definition. This makes computing def-use chains (the set of all reachable uses of a variable from its definition without any other definitions in-between) trivial. It also greatly simplifies liveness analysis (the range of instructions which make use of a given definition). Secondly whenever a new definition occurs, previous ones are dead. This makes register allocation almost trivial too, as any two definitions are independent and can thus use the same register if and only if their live ranges are disjoint.

$\phi - nodes$ **and** $\phi - functions$

Now that the principle and advantages of SSA are known, let us see implementation difficulties. While applying the principle is trivial for any purely linear program, complexity comes whenever it is not. Indeed, if a basic block is conditionally executed, the next block may have to choose between two copies of the same variable. To understand the problem, see Fig. 2. What can be seen there is a case where, after an if-

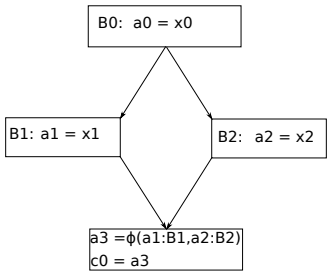


Figure 2: $\phi - functions$

else statement, variables in the final block have two possible values with two different names. This would obviously be a problem at run time. To mark this problem, which will later be solved by renaming all the values with the same name, a $\phi - function$ will be placed. This function is virtual (it is not really executed) and could be read as "a_3 takes value a_1 if coming from block 1 or a_2 if coming from block 2". In the rest of the report, a node containing a $\phi - function$ is called a $\phi - node$. The set of all nodes possibly containing a $\phi - function$ is the Iterated Dominance Frontier. Dominance referring to the portion of the code where one definition of a variable dominates its use, *i.e.* any use is linked to this definition. Dominance frontier is the point where this dominance ends. When two dominance frontiers are at the same point, it then means that two definitions of a variable are alive together. It is then necessary to insert a $\phi - function$ to choose the right definition and transfer it to a new copy. These nodes are only "possibly" ϕ as definitions inside may die before ever being used, thus removing the need of a $\phi - function$. Finally, when it is just created, SSA is in a from called Conventional SSA (CSSA). All copies of the same variable have non

overlapping live ranges, and any two copies in a $\phi - function$ in particular.

DJ-Graphs

One last useful notion for SSA is DJ-graphs. A DJ-graph is essentially a transformation of a CFG but with two types of edges and a stronger order. Each vertex has a level which depends on dominance. If a vertex is dominated by another, *i.e.* attaining this vertex from the root implies going through its dominator first, its level will be the level of this vertex + 1. At the same time an edge colored with a "D" will be drawn between them, it will be called a D-edge. A vertex which is targeted by a D edge will then systematically be lower in the graph than its dominator. As for the other type of edges, J-edges, they are drawn between two vertices in the case where it is possible to reach the second one from the root by passing through the first. A J-edge directed at a vertex implies that multiple path can lead to it. These graphs are very useful to computes SSA form in a faster way. Thanks to them, finding which node may be ϕ implies much less redundancy. Their use in SSA construction and $\phi - functions$ placement was first introduced in [Sreedhar and Gao, 1995].

3.2 Optimizations

When speaking about SSA, it is also important to discuss optimizations, as these are one of the main motivations behind its implementation.

Conservative Optimizations

The main characteristics of conservative optimizations is not to create interferences between variables which did not interfere before. A well known one is liveness analysis. It consists in an elimination of dead variables and instructions. Live ranges are first computed and then used to eliminate variables which are never used. In a regular compiler case, SSA aware liveness analysis is rarely used as live ranges are needed to form a pruned SSA form(without dead $\phi - functions$). Yet, in DBT, a formal liveness analysis is often need, or useful, as this step allows to determine which temporary variables should effectively be synced to memory(as a variable) at the end of the block. Using a SSA aware algorithm such as the one in [Boissinot *et al.*, 2008] then becomes quite useful, as it exposes multiple advantages compared to regular algorithms. In particular, it is faster and has vastly improved resistance to code transformation. This shows the interest of optimizations in SSA. Especially conservative ones.

Destructive Optimizations

Destructive Optimizations, as opposed to conservative ones, have for main characteristics to generate new interferences between variables. A perfect example of those is constant folding. When executed, this algorithm can drastically change the live range of variables. Indeed, it proceeds by eliminating variables which are copies of others, thus extending live range of the "primary" variables. It also suppresses instructions which were manipulating said copy or constants. After its pass, the SSA form will have been broken as variables which are part of a $\phi - functions$ are copies of each other. Thus algorithm will change their live range and have them interfere with each other and with copies generated by the function itself. Other optimizations such as code motion will have similar effect. An example of code motion effect can be found in figure 6, other examples may be found in [Sreedhar *et al.*, 1999]. Even though these optimizations can seem harmful, they are beneficial to generated code quality. It may then be useful to still use them when possible.

3.3 Coalescing

A last notion that should be discussed is coalescing. While it could be considered to be an optimization, it is presented separately as it is a mandatory step, and not optional like other optimizations are. It is the step in register allocation that consists in placing each variables in registers. Moreover, out of SSA algorithms are a special case of coalescing. The problem is similar to graph coloring and is thus NP-Complete. Yet, in SSA, the problem is actually simpler. Since every variables are defined once and only once, computing their live range is thus simpler and their value is known along their whole life. Following [Boissinot *et al.*, 2009] method to compute intersection, it is then possible to allocate registers by value and not just by variables. In general, live ranges being cleanly separated(only one definition) knowing for the exact length of register occupation by any variables is trivial. Coalescing itself is a fixed point algorithm that, given a set of variables, reduce it as near as possible to a set of independent variables. Independent variables being variables which can not be placed in the same register. The fact of imposing a limit in the resulting number of variables is the actual register allocation. These variables will then be *colored* with a register number. Obviously, it is not always possible to reach the desired number. There will then be *spilling* which means that registers will be put on the stack instead. The problem of choosing which register to spill, even in SSA, is still NP-Complete([Bouchez *et al.*, 2005]). Hopefully, in this work only *aggressive* coalescing is needed, *i.e* the number of color is not limited(see [Boissinot *et al.*, 2009]). It is used to exit from SSA form, which only means reducing as much as possible the number of variables yet without constraints on the desired number. Reaching optimal solution is still a fixed point problem. Still it can be bounded if an optimal solution is not the objective, as is the case in this work where speed is sought for, not code quality itself.

4 Related Works

The first and foremost work that should be observed is the one at the origin itself of DBT. The concept is, in fact, not recent. Its first traces can be found in the 80's in [Deutsch and Schiffman, 1984]. Even though it was not yet dynamic binary translation but dynamic translation in a more general sense, it had all the core ideas. Instead of generating code for a real machine directly, it was first generated for an abstract machine. At runtime, this abstract code would then be translated to native code whenever the block of code, or function, would need to be executed. Resulting code would then be cached for later use.

Now, for DBT and SSA, while to the best of the author's knowledge, no work about SSA IR for Qemu has been done, there are traces of such work in at least one other DBT en-

gine: crossbit [Yindong *et al.*, 2010]. But the authors designed their own simulator and thus could implement their own IR as they saw fit, even though they do not detail how they perform optimizations. This is one important difference with this work as the SSA here is to be implemented in an existing simulator. Compatibility with existing backends and frontends has to be kept, leading to other difficulties. Also, while with their most recent IR [Yang and Guan, 2012] they affirm being faster than Qemu and Valgrind, no code or binary seem to be readily available to test the current version of Qemu against them and, in the original article, they admit themselves that their simulator is not functional (it lacks proper exception support, ...) and it only seems to support application level DBT (not OSes). So, comparing an incomplete simulator, designed only to simulate user space applications with a full featured system level simulator raises a few questions, making it hard to base our work on their results. Anyhow, it is not possible to decide if it is useful or not to use an SSA IR using these works.

In the state of the art DBT such as Qemu [Bellard, 2005] or UQDBT[Probst, 2001], the IR is not in SSA. Only a few properties are expected from those. For example, the number of virtual registers (or temporaries) is considered infinite. This is enough to achieve the objective of reusable DBT without making frontends programing overly complex. LLVA, the IR from LLVM [Adve *et al.*, 2003] is also in SSA. It is also widely used and studied. Nonetheless, LLVM is not quite adapted to full system simulation. It was designed with C or C++ source code as an entry and not pure binary, even less an OS. Challenges are not quite the same.

In the same way, Jikes rvm [Alpern *et al.*, 2000] has similar problematics: it is a virtual machine, programs are translated dynamically, SSA is used for some optimizations [Van-Drunen and Hosking, 2004]. However, it is targeted for class files which gives much more information than what can be obtained from a binary (number of variables, functions, size of the code, standard frame size between others). In fact whole part of the optimization gain there will be related to inlining functions and procedures, which is very hard to do in DBT, and improving garbage collection which is not implemented in Qemu nor in any standard simulator to the best of our knowledge. While inlining could be an interesting addition to DBT, and quite probably lead to good performance increase, it also needs to rethink the generated code cache management system which would require an important amount of time and effort to be done.

For SSA itself, literature is quite large. This work focuses on algorithms to construct and destroy SSA form. In [Boissinot *et al.*, 2009] or [Sreedhar *et al.*, 1999] a focus is given to design general algorithm to get out of SSA in any situation, *i.e.* even when some properties have been violated by optimization pass. While the former algorithm claim to be usable in JIT (and thus probably in DBT), technical problems prevented us to use it. Indeed, the insertion of copy instructions during the optimization pass breaks the consistency of Qemu IR, thus rendering impossible the use of general algorithms. Instead of using this general algorithm, it was then preferred to remain in CSSA form of which renaming is enough to get out. This still opens the possibility to try to eliminate as many "copies" as possible, which needs to be evaluated to know if will be beneficial or not.

5 Contribution

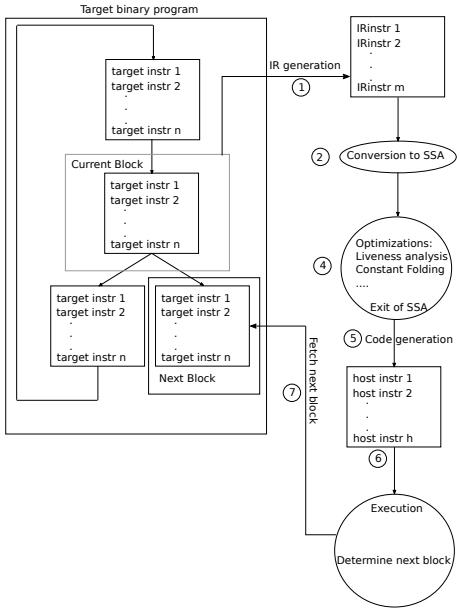


Figure 3: Code generation process with SSA

The main contribution of this work is the definition of a method to transform an intra basic block IR into an equivalent SSA form during DBT. Code generation in DBT is done block by block a in way that is oblivious to other blocks. Our SSA version of IR integrates in that phase, the SSA transformation is thus intra basic block. Figure 3 depict that process. As was explained in section 2.1, firstly instructions of the current block are processed one by one. After this phase, an IR block is obtained. It is in this IR block that SSA transformation take place. Optimizations are then executed. Once optimizations are done, it is important to get out of SSA, in particular to solve $\phi - functions$ problem. During this exit of SSA, optimizations are done too. Finally, after destroying the SSA form, host code generation takes place and the block can be executed, next block will be determined and then be translated if not already done, executed directly otherwise.

In order to achieve the implementation of an intra basic block IR in DBT, an extensive study of the problem had to be done. Amongst the questions were, what would be the processed code's control flow graph (CFG) form? How to then place ϕ-functions? Where could speed ups be expected? And also clearly, in order to experimentally be able to assess the interest of the proposal, what could be implemented or not into a DBT engine, *e.g.* Qemu in this particular case?

5.1 Preliminary work

Preliminary work mostly consists of an analysis of existing literature about SSA and understanding what is reasonable to implement in a DBT simulator. We first look into the case of

an ARM frontend. Backend for all this work is x86_64 (even though any backend should work). The first problem to tackle is to create the SSA IR form. In this creation, the costliest part is theoretically placing $\phi - nodes$, this complexity is however strongly related to the form of the CFG.
We also focused at first on a single processor, to identify the issues one by one. By analysing the ARM assembly language, one can guess the possible CFG of a regular DBT IR such as the one from Qemu. Indeed, the only instructions causing a branch but not a change of basic block are *predicated instructions*. These are instructions which execution depends on whether some flags are set or not. In the IR CFG, they then form a triangle as in Fig. 4. The presence of

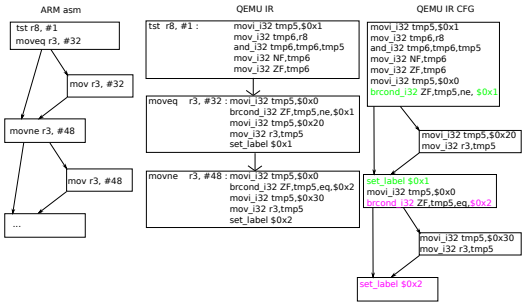


Figure 4: ARM assembly and its resulting IR and CFG

this form does mean that there will be $\phi - nodes$, but it also means that these should not be overly complex to compute. Only loops and other backward branches are actually problematic. Hopefully, none of them exists in ARM instruction set, except for these triangles, the CFG is a straight line.

5.2 Algorithms

Design

Knowing these restriction, one can get optimistic about constructing an SSA form, and in particular place $\phi - nodes$ in DBT simulators without major slow downs. The $\phi - node$ placing algorithm comes directly from [Sreedhar and Gao, 1995]. Firstly, a DJ-graph, is constructed. This graph contains two types of edges: J-edges and D-edges. A D-edge between two nodes indicates that the second node can *only* be accessed, from the root of the graph, by passing through the first one. A J-edge on the contrary correspond to an edge of the CFG which is optionally taken, *i.e.* it is *possible* to get to the second node by passing through the first one. A simple way to see things is to say that a node is part of the Iterated Dominance Frontier (IDF), meaning it is possibly a $\phi - node$, if there are one or more J-edges directed toward it, *i.e.* some of its direct predecessors are *conditionally* executed. In our case, thanks to the fact that we work on a basic block per basic block basis, this simple way is actually the exact way to compute the IDF. With the CFG being oriented in only one direction, a node may be ϕ if and only if a direct predecessor node has a J-edge toward it. From this observation, one can write the simplified Algorithm 1. It makes use of a dequeue data-structure to create a topologically sorted DJ-graph.

The DJ-graph creation, see Algorithm 1, proceeds as follows: First the nodes are computed, then they are used

Algorithm 1: DJ-graph computation

Data: IR basic-block (array of instruction) T
Result: A topologically sorted DJ labeled IR CFG $G(N,E)$ and the Iterated Dominance Frontier IDF

$N \leftarrow \emptyset$;
$n_0 \leftarrow$ new_node() // A node contains its first and last instruction plus a level and ϕ variable set if need be.
$n_0.first\_instruction \leftarrow T_0$;
for *i in 0 .. T.length* **do**
  if T_i = *branch* **then**
    $n_i.last\_instruction \leftarrow T_i$;
    $N \leftarrow N \cup n_i$;
    $n_{i+1} \leftarrow$ new_node();
    $n_{i+1}.first\_instruction \leftarrow T_{i+1}$;
    $n_{i+1}.last\_instruction \leftarrow T_{T_i.dest-1}$;
    $E \leftarrow E \cup (n_i, n_{i+1}, D)$;
    $E \leftarrow E \cup (n_i, n_{i+2}, D)$;
    $E \leftarrow E \cup (n_{i+1}, n_{i+2}, J)$;
    $n_{i+2}.isPhi \leftarrow true$;
    $IDF \leftarrow IDF \cup n_{i+2}$;
    $i \leftarrow T_i.dest$ // The next interesting instruction is the destination of the branch.
  end if
end for

to create and color the DJ graph. Nodes can be considered to be retrievable by the index of their last instruction. For each nodes only the first and last instructions are memorized so as to be able to go through instructions it covers later without having to actually store all of them. Creating D and J edges can be done as, at that time, all nodes are known in the CFG. D edges are directed to all successors as, without any backward jump in the block, it is impossible to find any path not going through the current node to its successors. J edges are between the predicated nodes and non predicated one as these are, obviously, optional paths. The visit function found in [Sreedhar and Gao, 1995] was not used, determining $\phi - nodes$ was done directly while constructing the CFG and DJ-Graph. Generality was sacrificed as the absence of loops means that there is no need to be careful, whenever a J edge is found, its target is inserted as being IDF. In the general case, this would hardly be possible as loops and backward jumps would then make it possible to duplicate nodes in the IDF and thus potentially $\phi - functions$ which could cause correctness problems. However, in this case, there are no loops or backward jumps, so the algorithm suffice in this form to guarantee that no node is inserted in the IDF for naught or is studied more than once. It is thus a better solution to build the IDF at the same time as the DJ Graph as it then becomes mostly costless. DBT being an environment where any seemingly slight performance loss may in practice slow down applications dramatically, any and all possible optimizations should be done.

The algorithm 2 proceeds through instructions to make variables in SSA form. Destination variables are replaced by

Algorithm 2: Conversion to SSA

Data: The IR block's CFG and its instruction set T
Result: All variables are in SSA form(only one definition per variable)
// All instructions have a destination where the result is stored and sources used for computations(destination may also be a source);
for *i in 0 .. T.length* **do**
  if AlreadyDefined(T_i.*argument_destination*) **then**
    T_i.*argument_destination* ← CreateCopy(T_i.*argument_destination*);
    T_i.*argument_destination.birth* ← T_i;
  end if
  foreach *Source s in* T_i.*argument_sources* **do**
    if *ThereExistCopy*(*s*) **then**
      s.last_use ← T_i;
      s ← LatestCopy(s);
    end if
  end foreach
end for

a new copy if they were previously defined. Then, to preserve consitency, sources are replaced by their latest copy if one exists. At the end, every variables of the IR is defined once and only once and each copies of a same varibles have disjoint live ranges. Consequently, the code is not only in SSA form, it is in the stronger CSSA form. Thanks to that fact, it is possible to use a simple Out of SSA algorithm on it, as long as no destructive optmizations are executed. At the same time as the form is constructed, live ranges of each variables are computed. These data will be used to compute live out sets in the $\phi - function$ placement algorithm 3. After executing this algorithm, the only step left to build SSA form is $\phi - functions$ placement.
While the IDF has already been computed, it would be sub obtimal to place $\phi - functions$ immediately. Some of them might in fact contain only dead variables. Thus, as an improvement to the method of [Sreedhar and Gao, 1995] which explained how to compute IDF in linear time, algorithm 3 was added. Looking at example 5, one can see that while J edges

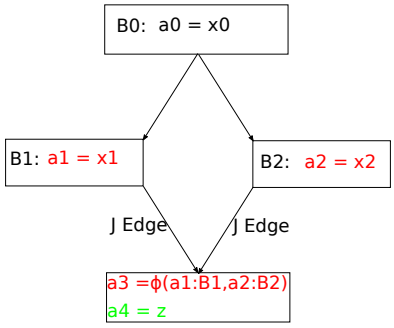


Figure 5: Interference of Dead Variables

are pointing at B, there is no need of a $\phi - function$. All concerned variables are actually dead, computing a $\phi - fucntion$ for them is thus a non sense as they will be eliminated by liveness analysis. Thus, algorithm 3 inserts $\phi - function$ only for live variables, hence avoiding this problem. This is done by checking, for each variables with multiple sources, whether they are used or not(if they are part of the live out set).

Algorithm 3: Phi-function insertion

Data: *IDF* The pre computed Iterated Dominance Frontier
Result: $\Phi(N,V)$ A set of nodes and their ϕ-functions
$N \leftarrow \emptyset$;
$V \leftarrow \emptyset$;
foreach *Node* $n \in IDF$ **do**
  foreach *Variable* $v \in n.PhiVariableSet$ **do**
    // The PhiVariableSet is the set of variables which began their life in a predecessor predicated block;
    if *liveOut(v)* **then**
      $N \leftarrow N \cup n$ // if n already there, just ignore;
      $V \leftarrow V \cup v$;
    end if
  end foreach
end foreach

The form which is obtained after executing copy insertion algorithm 2 and Algorithm 3 is still CSSA, as any copies of the same variable will not have overlapping live ranges.
Once in SSA form, and potential optimizations have been done, it is necessary to transform out of SSA. For now these optimizations are limited to a simple liveness analysis, no other optimization were done on the SSA IR. The part which was chosen for optimization is the out of SSA itself. What [Boissinot *et al.*, 2009] mainly taught us was that exiting SSA, when in CSSA form, essentially is an aggressive coalescing problem. Knowing that DBT frameworks creates a potentially large number of temporaries, eliminating a number of them should lead to reduced spilling on host side. It was thus decided to execute a simple, proof of concept, coalescing algorithm at SSA exit. Assuming primary variables are variables that were not added by SSA construction, algorithm 4 is obtained.

Algorithm 4 proceeds by going through each primary variables and finds the first one with which it will be merged. To decide whether or not to merge, the only test is to check that live ranges of both variables don't intersect. For now, this algorithm will, at best, reduce by half the number of variables and is quadratic with respect to the number of primary variables. It remains that this algorithm is not fully satisfactory and a better, already designed one will be presented in a later part of this section. Indeed, in a first time, this work focused on keeping simple algorithms that would fit well in our simulator. Adding copies, as stated in Section 4, currently breaks consistency in Qemu IR. This issue was not addressed as it would essentially be used only for a general out of SSA algorithm which did not yet have a use. The second grief is that coalescing can obviously be done in a better way. Even if the limitation of merging two

Algorithm 4: simple Coalescing and SSA exit algorithm

```
Data: V a set of CSSA variables
Result: V is no longer in SSA, first level disjoint
        variables were merged
foreach unmarked primary variables p ∈ V do
    foreach unmarked primary variables v ∈ V \ p do
        if live(v) ∩ live(p) = ∅ then
            mark p and mark v;
            rename v as p;
            break;
        end if
    end foreach
end foreach
foreach variables v ∈ V copy of primary p do
    rename v as p;
end foreach
```

variables at most was done, searching for the "best" match would probably enhance code generation more. A more complete set of algorithm could then be algorithms 5 and 6.

Algorithm 5: Out of SSA copy insertion, directly inspired by method II of [Sreedhar *et al.*, 1999]

```
Data: A set of SSA variables
Result: A set of CSAA variables
foreach φ − functions Φ do
    foreach Copy C_i ∈ Φ do
        foreach Copy C_j ∈ Φ with i ≠ j do
            if live(C_j) ∩ live(C_i) ≠ ∅ then
                Insert copies of C_i and C_j in their source
                block;
            end if
        end foreach
    end foreach
end foreach
```

Algorithm 5 aims at transforming a program in any state of SSA form back into a CSSA form. As some optimizations such as copy propagation may modify live ranges of variables, these might result in obtaining a "TSSA" form. This would mean a SSA form without guarantee that no copy of a same variable may have intersecting live range. Traditional, simple out of SSA algorithm then becomes unusable as there is a risk to rename variables which have different values at the same time, see figure 6. To avoid this, before proceeding to the renaming, copies are inserted for problematic variables. Variables will be considered problematic if, being related by $\phi - functions$, they have intersecting live ranges and may survive further than their respective $\phi - function$. In that case, copies will be added to make sure these variables are killed at the time of their $\phi - functions$. This algorithm overestimates the need of copy but is better than a naive approach. Still, it is correct which is the foremost interest of this work for now. This algorithm was not used by [Boissinot *et al.*, 2009] as it

Algorithm 6: Coalescing function

```
Data: A set of SSA variables V
Result: A minimal set of register allocatable variables
while There exist v1 and v2 such that live(v1) ∩ live(v2)
= ∅ do
    foreach Primary variables v ∈ V do
        bm ← FindBestMatch(v);
        rename v with bm;
    end foreach
end while
```

Algorithm 7: And find best match sub function

```
Data: A variable v
Result: Returns best match for variable v
min ← ∞;
bestMatch ← nil;
foreach primary unmarked variables v2 ∈ V \ v do
    S ← Coalescing(V \ {v,v2});
    if S.length ¡ min then
        min ← S.length;
        bestMatch ← v2;
    end if
end foreach
return bestMatch;
```

implied for them to compute interference graphs. Yet, Qemu and most likely other DBT simulators need to compute liveness of variables, even in SSA form, and as such, liveness information are obtained in a costless way. Reducing even slightly the number of copies can help for final renaming and as such it seems beneficial to use it.

Algorithm 6 is a stabilizing algorithm. It proceeds by going through the set of primary variables, find best pairs to be renamed and then pass again for as long as variables which can be renamed together exists. To find the best match for any variable, it re-execute itself with the variable set minus considered pair and compare size of resulting sets. It should be noted that here, the live range of a primary variable is considered to span from its definition to the death its last *copy*. The solution implicitly does correctness wise what is done in [Boissinot *et al.*, 2009]. Merge is done for primary variables and all their copies at the same time and as such, for all ϕ congruence classes, without any internal live range checking. There is no problem with variables inside a ϕ congruence class either as 5 execution ensure that we are in CSSA form once again. The out of SSA itself has implicit correctness as two variables are renamed with same name if and only if none of their copies have intersecting live ranges. Optimization itself is slightly weaker as variables are not merged by values but only with respect to their live range. Some copies which could have been avoided with Method III of [Sreedhar *et al.*, 1999] will not be eliminated either as constraint on live ranges are stronger here.

Once implemented, algorithm 6 should allow to get better runtime performance, furthermore if implemented in an inter

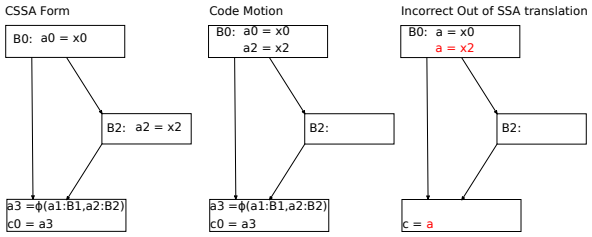


Figure 6: Erroneous out of SSA translation

basic block fashion. However, inter basic block optimization pauses the problem of cache management which does not allow to change already generated code and thus to replace previously generated block by an optimized complete code path. Still, most implemented algorithms would be able to support such a case with little to no modification, only exiting SSA needs to be made generic using algorithm 5 as a pre pass so as to avoid live range problems.

6 Experimental Validation

6.1 Performances

Protocol

On the platform side, benchmarks were ran on a linux server. The distribution that was used is debian Jessie. Hardware that was used is 32 Xeon Cores @ 1,84GHz with 32 GB of memory.

All tests were ran with only one user on the server so as to avoid any impromptu context switches and any other kind of noise in the measurement. Pursuing the same objective, never more than 20 tests were executed in parallel and no other tasks were ran either. Both SSA and non SSA benchmarks were launched in parallel so that the server was in the same state for both runs. All cross-executed benchmarks were run bare metal, that is, without an operating system. They were cross compiled to ARM with `gcc` O2 level optimization and then loaded as a kernel in Qemu.

To obtain statistically valid results, every performance related tests were executed 120 times. The average value was taken to be used in graphs with its standard deviation.

Only for cache misses, where cachegrind was used and as this tool is deterministic, tests were ran only once.

As for Qemu configuration, the platform that was used is *integratorcp*. The only modification done to the platfrom itself is an exit trap (write at an address to shut Qemu down). With this platform, the standard quantity of memory was used, that is 128MB.

The performances were evaluated using the linear algebra set of the Polybench benchmark [Pouchet, 2012]. This set of tests was chosen because it is representative of multiple interesting cases. On one side, programs which are short and don't go through the same code many times. Those have execution time which are largely dependant of compilation time. On the other side, long running program going through the same loop over and over. These have execution time which mainly depends of generated code performance. They also expose different sizes of blocks and number of variables. These elements are even more interesting than run time.

Run time

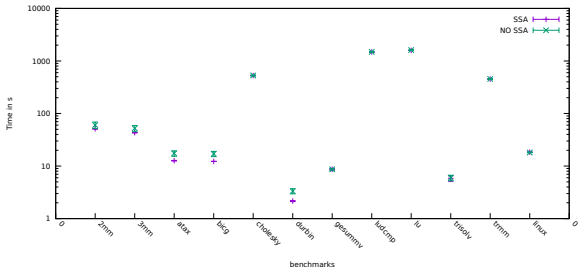


Figure 7: Run times on various benchmarks

On Figure 7 one can see the run time of Qemu with and without SSA. It can be seen that there is a performance gain when using SSA. However, this gain seem to be variating depending on the benchmarks. On one hand, benchmark 2mm is 9 seconds faster with SSA. On the other hand, ludcmp runs at about the same speed (both results are contained in the error margin of the other). 3mm is also about 9 seconds faster on average but gesummv ran at about the same speed in both cases. In all cases, Qemu without SSA exposed important error margins. There was overall a performance gain, it was however not uniform. In an attempt to explain it, an instrumentation of the code was done in order to determine what time was spend in optimization and translation in general. The results can be seen in Figure 8. For all chosen bech-

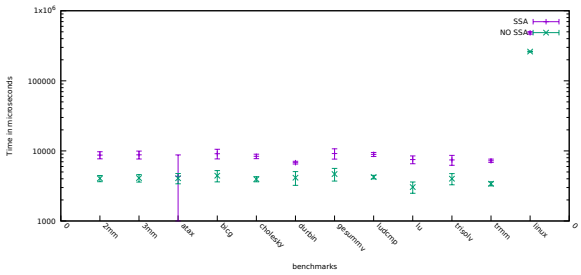


Figure 8: Time spent in Qemu middle end/backend for one execution

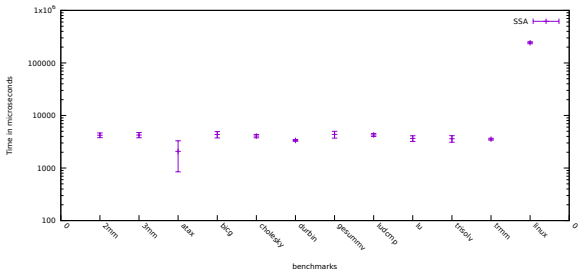


Figure 9: Time spent going in and out of SSA

marks, time spent in middle and backend never really exceeds half a second, even on a linux boot which is a worst case for translation. In cholesky case for example, 0,003 seconds are spent for non SSA version of Qemu and 0,008 for SSA version. Running linux, time spent in backend manage to reach around 0,48 seconds with SSA. This means that any observed performance gain, or its absence, can hardly be related in any way to a variation in translation time. The observation of Figure 9 shows that both conversing to SSA and going out of it, even when both are summed, remains a low overhead. Compared to execution times seen in Figure 7, translation times and even more so SSA conversion and destruction are clearly negligible. Therefore, one can conclude that SSA didn't bring any significant overhead, it is about 1,5% even for linux, and yet brought significant speed ups on multiple benchmarks. Furthermore, it shows that there is still room for optimization. As long as the run time performance gain is large enough, slowing down translation further will not be a problem. Even if it came to reach 2 or 3% of overhead, the run time gain would remain superior.

Standard Deviation

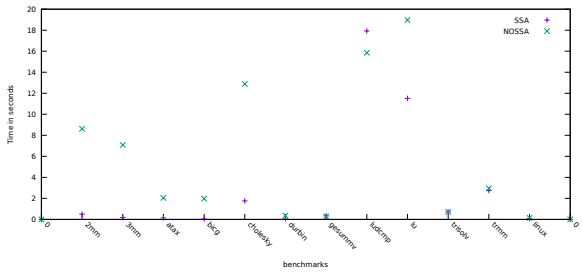


Figure 10: Standard deviation on various benchmarks

As observed in previous subsection, Qemu without SSA has a large standard deviation. It is another interesting phenomenon to be observed in this work, in most benchmarks, there is an important reduction of standard deviation when using SSA IR. Indeed, as one can see in figure 10, for most benchmarks that were used in these experiments, standard deviation was sensibly reduced compared to Qemu without the SSA IR. It means that an element of performance which tends to vary greatly otherwise gets more stable. For example, on the benchmark cholesky, while the standard deviation reaches 12,89 seconds for Qemu without SSA, it goes down to 1,76 seconds when SSA is used. As for atax, it is reduced from 2,05 to 0,15. For atax is also interesting to note that standard deviation of SSA translation time itself was actually high. Linux exposes next to no effect on standard deviation. Just as there was mostly no effect on performances. Only ludcmp really figures as an exception here, with 17,93 seconds standard deviation with SSA against 15,86 without it. It is possible that some specificities of this benchmark render SSA implementation less efficient. While further investigation is needed to give a definitive answer, the most likely reason would be that most variables in this program have intersecting live ranges. Probably Linux presenting no reaction to SSA implementation is explained in the same way. Consequently, Out Of SSA algorithm would then recreate a program nearly identical to the one before SSA pass. In any other tested programs, none of the decreases in standard deviation are negligible. Undeniably, implementation of SSA into Qemu had an effect on variability in execution time. Parameters that may have execution time vary are usually related to memory accesses. This could, for example, be the result of better register handling. Indeed, register spilling implies a larger number of memory accesses and memory access latency, especially on an Intel host, can have a high variability. This variability being due to the complex memory hierarchy, associated with advanced prefetching and eviction algorithms.

Analysis

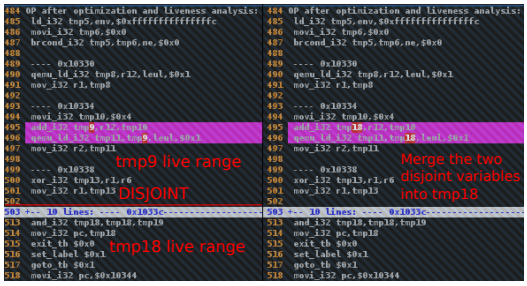


Figure 11: variable merging

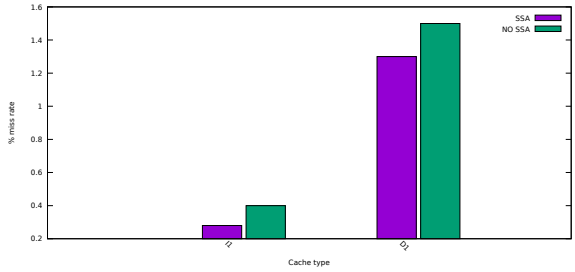


Figure 12: Instruction and data L1 Cache misses rate

Going back on what was observed before, it is quite safe to assume that the performance gain is related to a generated code run time improvement. Knowing that the only modification that was done on generated code is a reduction of the number of variables, one can deduce that spilling was reduced. An example of gain in number of variables can be found in 11. In that figure is represented a case where coalescing replaced a variable with another, disjoint one. This is akin to a register allocation, the two variables were disjoint and thus could be placed in the same register. Such an operation could be done until all variables left have overlapping live ranges and are impossible to place in one same register. Reducing the number of variable is in itself a way to use host registers in a more efficient way. Intuitively, as register allocation implies to map a set of variables to a fixed number of registers, having less variables should yield better results using any naive algorithm. The effect of this reduction in number of variables find its root in basic memory wall issue

as could be suspected with previous figures. When less variables are used, register are allocated in a better way and thus less memory accesses, in particular on the stack, are done. A measure of this effect in the case of Qemu is exposed in figure 12. There, it can be seen that the percentage of L1 cache misses, both on instruction and data side, has been sensibly reduced by the use of SSA. It goes down from 0,40% without SSA to 0,28% with it for instructions, and from 1,5% to 1,3% for data in the case of a Linux boot. While these numbers may seem to be negligible, even a slight percentage of cache misses may slow down program sensibly as these are costly. Especially, an instruction cache miss will unavoidably provoke slow downs as unknown instructions tend to stall pipelines completely. All of these results tend to confirm the importance of using coalescing at the exit of SSA and comfort us in thinking that improving further this algorithm is an important factor in further improving overall qemu performance.
Moreover, study of time spent in backend shows that translation time is mostly negligible in front of run time, even on such a complex program as Linux. From that observation, it becomes clear that an important margin for middle end optimizations exists. Which tends to prove the viability of the SSA IR approach to optimize Qemu code generation.

7 Future Work

While this work already managed to produce interesting results, we would like to improve it further. In a short term vision, we would like to improve the intra basic block performance. To this end, a first step will be to implement the algorithms that were presented in the contribution chapter. Finalizing coalescing in particular seems to be the best way to proceed. Seeing also that translation time does not seem a limiting factor for now, implementing some state of the art SSA optimization may be a good option. To have that, the general out of SSA algorithm detailed in the Section 5(Contribution) will be essential. Overall, short term work will mainly focus in improving existing SSA form in an intra basic block way.
For the longer term, multiple tasks will have to be tackled. While a natural continuation of this work would be to implement SSA in an inter basic block fashion, doing so require some deep modification into Qemu and most other DBT engines. Indeed, modification of generated code is complicated as of now. Due to the fact that in DBT, Unlike in *e.g.* java, the size of blocks, their number of variables, and in general the size of a code frame are unknown a priori, it is not possible to easily implement a modifiable code cache. This, however, is needed if we are to achieve a good performance boost with inter basic block optimization, as most optimizations achieving important performance gains would be destructive to the cache. Inlining for example requires to change the way the cache is addressed by current DBT so as to preserve its coherence. And yet, it would give an opportunity of important performance gain. This leads us to think that improving cache management in DBT is actually the next step in our work. The scheme should remain simple enough not to cause overhead, and yet respect the specification needed for efficient inter basic block optimizations. Once this would be done, implementing these optimizations would require to add target code profiling. While this has already been studied before, our approach would be novel as we would like to add efficient target code profiling and optimization in the case of multi processor target and host machine. n addition to this, another main problem nowadays in profiling is that it is more reactive than proactive. When defining a hot path, the decision need to be made early as explained in [Duesterwald and Bala, 2000]. Failing to do so will lead to miss the gain that optimized generated code could have provided. We propose to improve profiling of hot paths by using branch prediction, using [Faravelon *et al.*, 2015], to predict whenever branches are likely to be taken and thus whether the path will be just warm or hot. By predicting hot paths, while some optimization might be done for naught, real hot paths should be optimized sooner, thus improving the gain related to their optimization. To summarize, our long term target is to implement efficient generated code cache management along with profiling and optimization for multi CPU target machine with proactive approach to profiling. All of this being done while obtaining a good trade off between time spent in translation and run time performance gain.

8 Conclusion

In the end, this work has led to satisfying results. As a first contribution, in this report we proved that it was possible to implement SSA in a DBT simulator. Overall cost of this implementation was shown to be pretty much negligible in term of translation time. The proof of concept implementation that was done in Qemu slowed translation down slightly. Yet, overall runtime was in most benchmarks improved or stayed the same. And that, even though the implementation is still sub optimal. To this day, even on a complex program such as Linux, featuring self modifying code which can slow down sensibly Qemu code generation, translation time remains inferior to 2,25% of overall execution time. Execution time on the other hand was sometime improved by almost 20%. One can then conclude that not only SSA IR is a viable path to optimize Qemu code generation, but that there is still a significant margin to optimize.
This margin may be used to implement other SSA specific optimization and in particular the second part of our contribution. That is, the DBT usable algorithms that were provided in this report. In particular, a generic out of SSA algorithm may, in the long run, be an important addition to current implementation. Coupled with a more evolved coalescing algorithm such as the one presented in this report, generated host code performances could be significantly improved.
Implementing SSA in a DBT simulator did prove to be a difficult task. Little to no directly related work exists to this day. Even though extensive literature about SSA can be found, none of these works were done with DBT in mind. As a matter fact, the nearest works nowadays are targeted at Java-like virtual machines JITs. While these works are very interesting and can be inspiring, constraints are not the same. Thus the need to develop an approach specifically adapted to DBT.

References

[Adve *et al.*, 2003] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. Llva: A low-level virtual instruction set architecture. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 205, 2003.

[Alpern *et al.*, 2000] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[Bellard, 2005] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[Boissinot *et al.*, 2008] Benoit Boissinot, Sebastian Hack, Daniel Grund, et al. Fast liveness checking for ssa-form programs. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 35–44. ACM, 2008.

[Boissinot *et al.*, 2009] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoît Dupont De Dinechin, and Christophe Guillon. Revisiting out-of-ssa translation for correctness, code quality and efficiency. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–125. IEEE Computer Society, 2009.

[Bouchez *et al.*, 2005] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under ssa. Technical Report LIP-2005-33, Laboratoire de l'Informatique du Paralllisme, cole Normal Suprieure de Lyon, August 2005.

[Deutsch and Schiffman, 1984] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302, 1984.

[Duesterwald and Bala, 2000] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, 2000.

[Faravelon *et al.*, 2015] Antoine Faravelon, Nicolas Fournel, and Frédéric Pétrot. Fast and accurate branch predictor simulation. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 317–320. IEEE, 2015.

[Gligor *et al.*, 2009] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible mpsoc simulation. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 71–80. ACM, 2009.

[Pouchet, 2012] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite, 2012.

[Probst, 2001] Mark Probst. Fast machine-adaptable dynamic binary translation. In *Proceedings of the Workshop on Binary Translation*, volume 9. Citeseer, 2001.

[Sreedhar and Gao, 1995] Vugranam C Sreedhar and Guang R Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 62–73. ACM, 1995.

[Sreedhar *et al.*, 1999] Vugranam C Sreedhar, Roy Dz-Ching Ju, David M Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Static Analysis*, pages 194–210. Springer, 1999.

[VanDrunen and Hosking, 2004] Thomas VanDrunen and Antony L Hosking. Anticipation-based partial redundancy elimination for static single assignment form. *Software: Practice and Experience*, 34(15):1413–1439, 2004.

[Yang and Guan, 2012] Yin-dong Yang and Hai-bing Guan. An efficient adapting virtual intermediate instruction set towards optimized dynamic binary translator (dbt) system. *Journal of Central South University*, 19:3118–3128, 2012.

[Yindong *et al.*, 2010] Yang Yindong, Guan Haibing, Zhu Erzhou, Yang Hongbo, and Liu Bo. Crossbit: a multi-sources and multi-targets dbt. In *CLOUD COMPUTING 2010, The First International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 41–47, 2010.

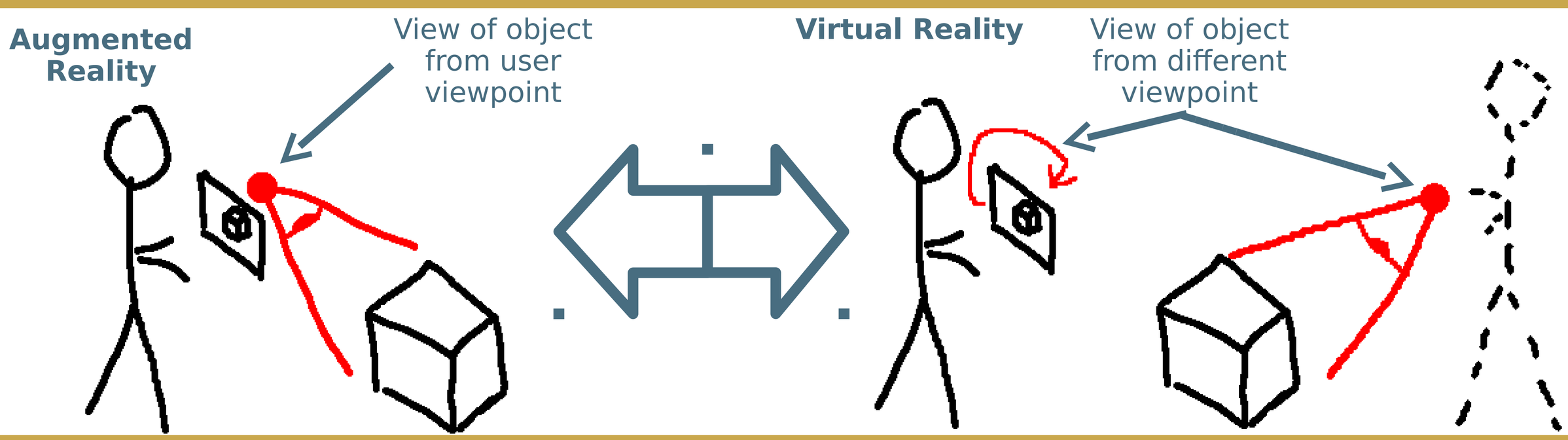
Interactive techniques for handheld augmented reality: switching between augmented reality and virtual reality

Valentin Prulière, Etienne Viallet, Laurence Nigay

IIHM, LIG, Grenoble, France;

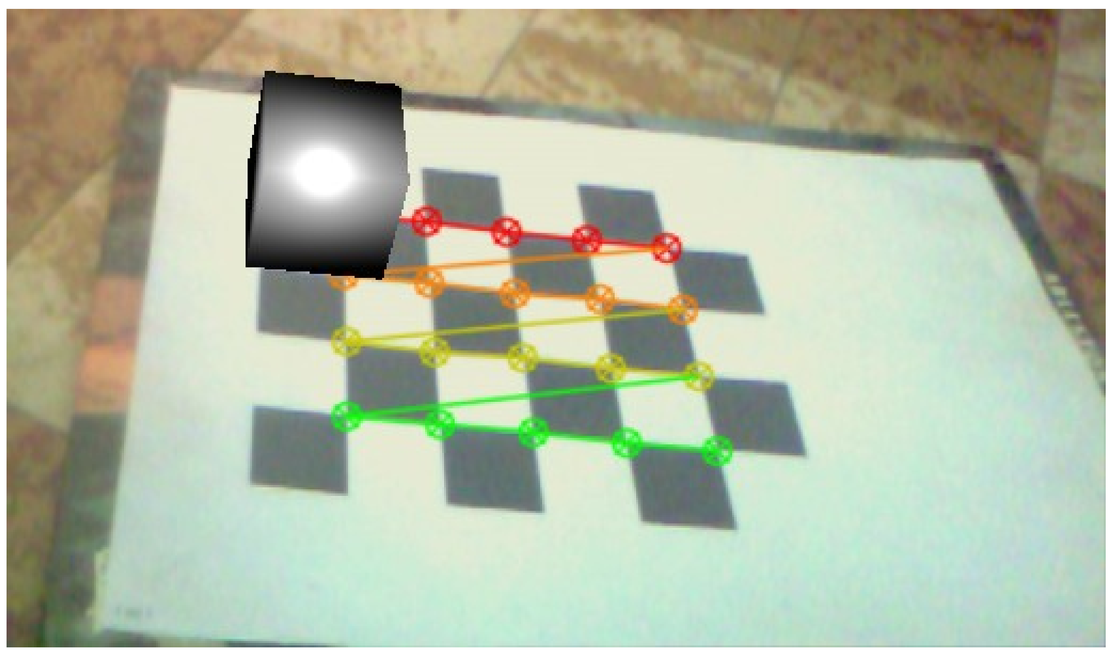
Using a mixed environment of Augmented reality and Virtual reality to assist in studying objects.
Study of techniques to avoid being desoriented within the environment.

Design



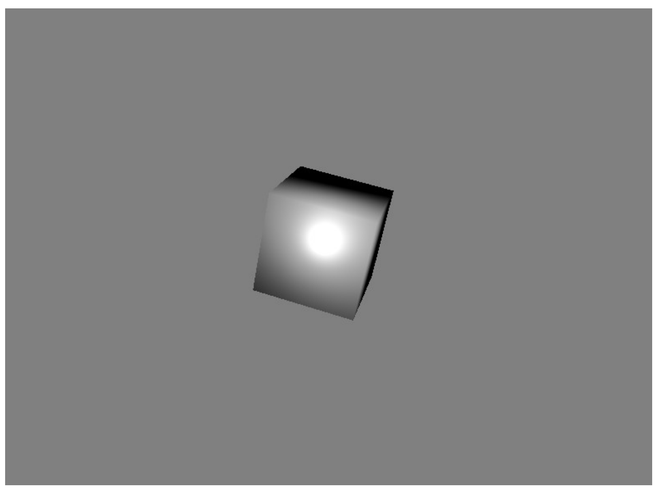
Interaction

AR view

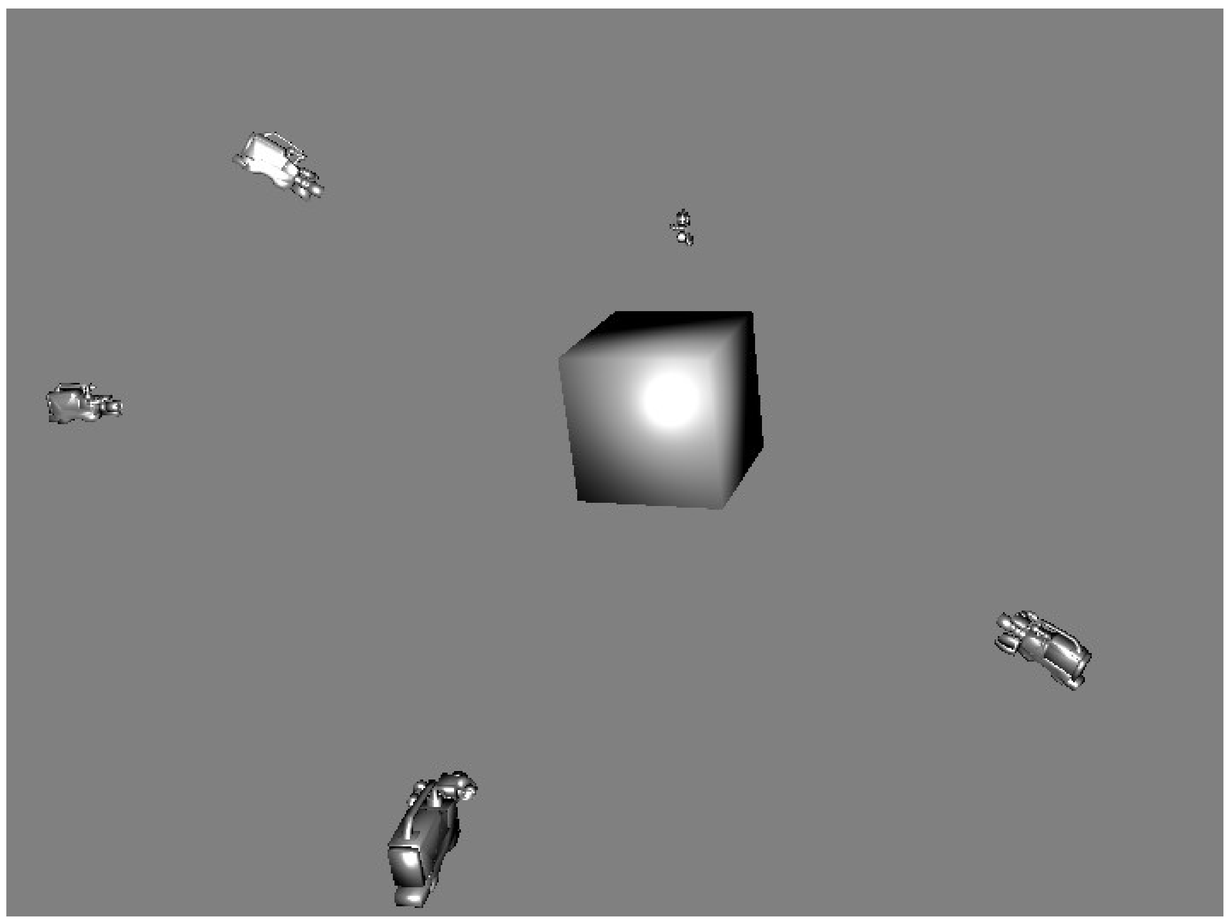


The view on the object as if the user were physically looking at it, with no change in the point of view

VR view



As soon as we leave AR and enter VR, only the 3D model is visible



Different points of interest are placed

pruliere.pdf: en attente d'inclusion

Finding Buffer Overflows Generating Loops

Claude Goubet 28/08/15

What is a buffer overflow?

A buffer overflow appears when a program, writing data in a buffer, overruns its boundary and overwrites the adjacent memory locations. This can lead to a software crash or allow access to forbidden memory locations and arbitrary code execution.

The string functions from C libraries are very vulnerable to buffer overflows.

```
strcpy(char *s1, const char *s2)
{
  char *s = s1;
  while ((*s++ = *s2++) != 0)
    ;
  return (s1);
}
```

Figure 1: C code of the strcpy function

Figure 1 illustrates the functioning of the strcpy() function.
The constant s2 represents the input string.
The variable s1 is the output string, copy of s2.
Strcpy will process all the the s2 string until the character pointed to by s2 is '\0', without checking s1's size.

```
.While:
    movl    -4(%ebp), %eax
    movzbl  (%eax), %edx
    movl    -8(%ebp), %eax
    movb    %dl, (%eax)
    movl    -8(%ebp), %eax
    movzbl  (%eax), %eax
    testb   %al, %al
    setne   %al
    addl    $1, -8(%ebp)
    addl    $1, -4(%ebp)
    testb   %al, %al
    jne     .While
```

Figure 2: x86 Code of the strcpy functions loop

How can a buffer overflow happen using strcpy()?

The figure 3 represents the organisation of the stack during a call of strcpy(). On the top of the stack are the variables used by the called function. At the bottom are the buffers containing the s2 string, the string to copy, and a buffer of free space allocated to s1.

Lets study the x86 code in figure 2:

Red : the memory write on s1
Brown : test if current character is '\0'
Purple : increment pointed address in stack
Green : back to the top of the loop
Both pointors are incremented until reach '\0'. If s1's buffer is smaller than s2 there is an overflow. It can then overwrite all the way to the return address.

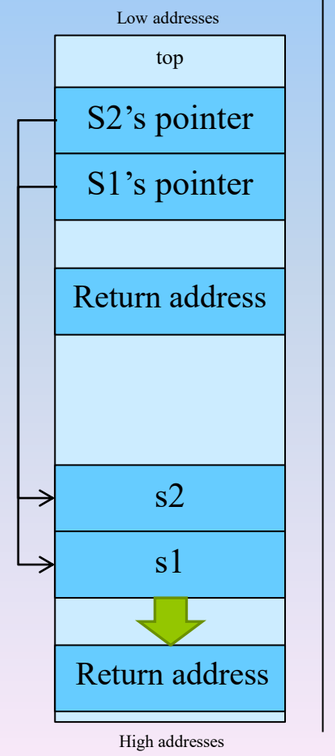


Figure 3: Stack representation in a strcpy call

How to detect a buffer overflow?

Idea : finding buffer overflows by statically detecting loops containing addresses which are modified at each iteration and checking if at least one of these addresses is written on.

First step: finding loops

From the control flow graph of the disassembled x86 version of a program (Figure 4.a) we create a domination tree in order to detect the back-edges (in purple in Figure 4.b). A loop will contain all nodes from the header to the back-edge node.
We must detect inner-loops (Green) and nested loops (blue) as well as nesting loops (Red).

```
int main () {
 int x = 0;
 while (x != 1){
  if (x < 1)
   x++;
  else
   x--;
 }
 while (x <= 15){
  int i = 0 ;
  while (i < 3){
   x++;
   i++;
  }
 }
 return 0;
}
```

Figure 4.a: example of C code containing loops

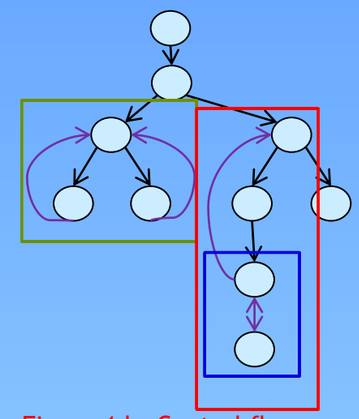


Figure 4.b: Control flow graph of Figure 4.a

Second step: Dataflow analysis

The dataflow analysis will allow us to know the state of every variable in term of data dependencies in the program and update the control flow graph. This way, a new edge will be set from each definition instruction to the definition instruction of its dependencies.

Detection

To detect the dangerous loops we can now enumerate the loops, climb up the data dependencies and find out if there is a call to a memory write instruction on a self-dependent memory address.
In the example of Figure 2, the red instruction would be detected and we would find that it writes on eax which depends on ebp (blue), when ebp is incremented in the purple section. Since an incrementing instruction leads to a self-dependency, this loop would be set as a vulnerable loop.

Finding Buffer Overflows Generating Loops

Claude Goubet 28/08/15

What is a buffer overflow?

A buffer overflow appears when a program, writing data in a buffer, overruns its boundary and overwrites the adjacent memory locations. This can lead to a software crash or allow access to forbidden memory locations and arbitrary code execution.

The string functions from C libraries are very vulnerable to buffer overflows.

```
strcpy(char *s1, const char *s2)
{
  char *s = s1;
  while ((*s++ = *s2++) != 0)
    ;
  return (s1);
}
```

Figure 1: C code of the strcpy function

Figure 1 illustrates the functioning of the strcpy() function.
The constant s2 represents the input string.
The variable s1 is the output string, copy of s2.
Strcpy will process all the the s2 string until the character pointed to by s2 is '\0', without checking s1's size.

```
.While:
    movl    -4(%ebp), %eax
    movzbl  (%eax), %edx
    movl    -8(%ebp), %eax
    movb    %dl, (%eax)
    movl    -8(%ebp), %eax
    movzbl  (%eax), %eax
    testb   %al, %al
    setne   %al
    addl    $1, -8(%ebp)
    addl    $1, -4(%ebp)
    testb   %al, %al
    jne     .While
```

Figure 2: x86 Code of the strcpy functions loop

How can a buffer overflow happen using strcpy()?

The figure 3 represents the organisation of the stack during a call of strcpy(). On the top of the stack are the variables used by the called function. At the bottom are the buffers containing the s2 string, the string to copy, and a buffer of free space allocated to s1.

Lets study the x86 code in figure 2:

Red : the memory write on s1
Brown : test if current character is '\0'
Purple : increment pointed address in stack
Green : back to the top of the loop
Both pointors are incremented until reach '\0'. If s1's buffer is smaller than s2 there is an overflow. It can then overwrite all the way to the return address.

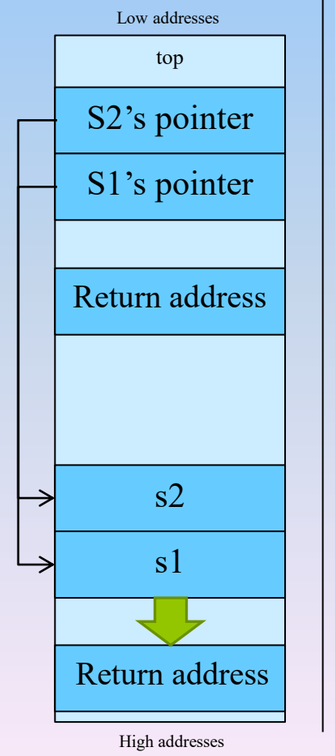


Figure 3: Stack representation in a strcpy call

How to detect a buffer overflow?

Idea : finding buffer overflows by statically detecting loops containing addresses which are modified at each iteration and checking if at least one of these addresses is written on.

First step: finding loops

From the control flow graph of the disassembled x86 version of a program (Figure 4.a) we create a domination tree in order to detect the back-edges (in purple in Figure 4.b). A loop will contain all nodes from the header to the back-edge node.
We must detect inner-loops (Green) and nested loops (blue) as well as nesting loops (Red).

```
int main () {
 int x = 0;
 while (x != 1){
  if (x < 1)
   x++;
  else
   x--;
 }
 while (x <= 15){
  int i = 0 ;
  while (i < 3){
   x++;
   i++;
  }
 }
 return 0;
}
```

Figure 4.a: example of C code containing loops

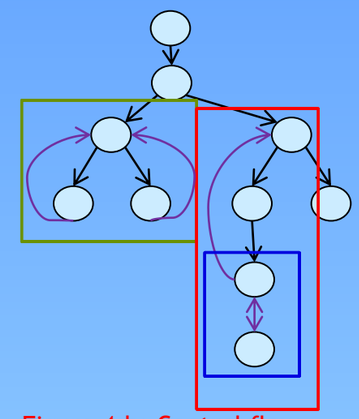


Figure 4.b: Control flow graph of Figure 4.a

Second step: Dataflow analysis

The dataflow analysis will allow us to know the state of every variable in term of data dependencies in the program and update the control flow graph. This way, a new edge will be set from each definition instruction to the definition instruction of its dependencies.

Detection

To detect the dangerous loops we can now enumerate the loops, climb up the data dependencies and find out if there is a call to a memory write instruction on a self-dependent memory address.
In the example of Figure 2, the red instruction would be detected and we would find that it writes on eax which depends on ebp (blue), when ebp is incremented in the purple section. Since an incrementing instruction leads to a self-dependency, this loop would be set as a vulnerable loop.